GUARANTEEING REACTIVE MISSIONS FOR COMPLEX ROBOTIC SYSTEMS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Jonathan Anthony DeCastro

May 2017

GUARANTEEING REACTIVE MISSIONS FOR COMPLEX ROBOTIC SYSTEMS

Jonathan Anthony DeCastro, Ph.D.

Cornell University 2017

With the availability of robots capable of performing complex missions, formal approaches to controller synthesis are gaining increasing attention as a means for synthesizing controllers that guarantee, by construction, the execution of such missions. The approach frees users from having to hand-design such implementations and have utility for missions in which the system must maintain safety in dynamic, uncontrolled environments. For instance, a personal assistant might be entrusted to fetch a box from a location and then deliver it to the required location in the household, while avoiding collisions with closed doors, people, and static obstacles. In this work, three interrelated problems are solved, each addressing the overall problem of automatically synthesizing controllers with applicability to robots with complex, nonlinear dynamics.

First, an off-line approach is introduced for synthesizing low-level controllers for nonlinear systems. The approach builds on sample-based motion planning and nonlinear system analysis techniques, and is capable of synthesizing a palette of controllers that certifies a task (mission instructions provided by a user via a formal specification). To address cases where a task cannot be guaranteed on a given physical platform, the second contribution is an automated approach to specification revisions using information contained in a discrete representation (i.e. abstraction) of the dynamical system. The user is provided with a concise yet expressive set of revisions as instructive feedback required

to guarantee the specification. The final contribution is a novel synthesis approach that extends the revisions approach to enable a team of robots operating in workspaces shared with other agents (e.g., humans or cars). Whereas typical centralized controllers are combinatorially expensive to compute, our approach lessens this complexity while retaining correctness by leveraging a local motion planner and employing a high-level scheme that reasons about deadlock between two or more agents. Throughout this work, the various approaches are demonstrated through a variety of missions carried out both in simulation and on physical platforms in dynamic environments.

BIOGRAPHICAL SKETCH

Jonathan A. DeCastro was born in Rochester, New York in 1977. He received his Bachelor of Science degree in Mechanical Engineering from Virginia Tech in 2001 and went on to earn a Master of Science from the same institution in 2003. He was employed as an aerospace researcher at NASA between 2004 and 2008. He subsequently began studies at Cornell to pursue a Ph.D. degree in Mechanical Engineering with minor concentrations in Computer Science and Computational Science and Engineering, which he earned in 2017.

For Joshua and Kelsey

ACKNOWLEDGEMENTS

First and foremost, I thank my family and loved ones, particularly my Mother and Father, Cheryl, and of course the little ones – Kelsey and Joshua – for their unending support and encouragement throughout this journey of mine. I deeply thank them for supporting me in my initial decision to enter grad school in the first place and just for being there during the ups and downs.

I thank all those at Cornell for their support: Hadas Kress-Gazit for her encouragement, constructive critique, and as a patient listener to my ideas; Ross Knepper for passing along his deep robotics knowledge, which has enabled me to accomplish a great many things, and Alex Vladimirsky for his input on the theoretical aspects of this work. I will always look back fondly to the camaraderie with the members of the Verifiable Robotics Research Group and the Autonomous Systems Lab – to everyone: I owe you a big thanks. Last (but not least), I am indebted to Marcia Sawyer for her meticulous guidance from start to finish.

Beyond Cornell, I thank the faculty and students of the NSF Expeditions from the Computer Augmented Program Engineering (ExCAPE) Project for imparting me with their deep formal methods knowledge and insightful discussions gained throughout the years of meetings, summer schools, and informal outings. In this realm, a special thanks goes to Paulo Tabuada, Lydia Kavraki, Ruediger Ehlers, Vasu Raman, Matthias Rungger, Ayca Balkan, and Salar Moarref. I further wish to thank Daniela Rus for sparking the multi-agent synthesis collaboration, and to Javier Alonso-Mora for the many interesting discussions and late-night coding sessions. Finally, I owe a debt of gratitude to Russ Tedrake and Ani Majumdar, with whom I had many insightful discussions with the funnels approach that were the genesis of many subsequent ideas. Without you all,

my graduate experience would have been far less of an enriching experience than it has been.

This work was supported by the NSF Expeditions in Computing project ExCAPE: Expeditions in Computer Augmented Program Engineering, grant number CCF-1138996.

TABLE OF CONTENTS

Biographical Sketch . . . iii
Dedication . . . iv
Acknowledgements . . . v
Table of Contents . . . vii
List of Tables . . . x
List of Figures . . . xi

1 Introduction **1**
1.1 Related Work Overview . . . 4

2 Synthesis of Nonlinear Continuous Controllers for Verifiably-Correct High-Level, Reactive Behaviors **8**
2.1 Introduction . . . 8
2.1.1 Problem Statement . . . 11
2.1.2 Chapter Outline . . . 14
2.2 Preliminaries . . . 15
2.2.1 Robot and Environment Abstractions . . . 15
2.2.2 Controller Finite-State Machine . . . 16
2.2.3 Continuous Dynamics and Operations . . . 17
2.3 Controller Synthesis Approach . . . 18
2.3.1 Composition Strategies . . . 18
2.3.2 Classes of Atomic Controllers . . . 20
2.3.3 Atomic Controller Synthesis Algorithm . . . 24
2.4 Computing Atomic Controllers . . . 28
2.4.1 Trajectory Generation . . . 29
2.4.2 Trajectory-Stabilizing Controllers . . . 30
2.4.3 Invariant Funnels . . . 30
2.4.4 Algorithm . . . 33
2.4.5 Complexity . . . 36
2.5 Controller Execution . . . 36
2.6 Simulations . . . 39
2.6.1 Patrolling Two Regions . . . 40
2.6.2 Pursuit-Evasion Game . . . 44
2.6.3 Delivery in a Cluttered Environment . . . 45
2.7 Conclusion . . . 48

3 Automated Generation of Dynamics-Based Runtime Certificates for High-Level Control **51**
3.1 Introduction . . . 51
3.1.1 Related Work . . . 54
3.1.2 Outline . . . 57
3.2 Preliminaries . . . 57

3.2.1 Linear Temporal Logic . . . 57
3.2.2 Discrete Abstractions . . . 58
3.2.3 Controller Synthesis . . . 60
3.3 Problem Formulation . . . 64
3.4 Revising Unrealizable Specifications via Counterstrategies . . . 67
3.4.1 Preventing Deadlock . . . 71
3.4.2 Preventing Livelock . . . 81
3.5 Creation of Runtime Certificates . . . 88
3.5.1 Parsing the Revisions . . . 89
3.5.2 Graphical Visualization Tool . . . 92
3.6 Case Studies . . . 94
3.6.1 Abstractions . . . 94
3.6.2 Revisions in a Finely-Partitioned, Temporal Abstraction . 95
3.6.3 Workspace Re-Partitioning in the Activation/Completion Paradigm . . . 98
3.7 Conclusions . . . 101

4 Multi-Robot Reactive Mission and Motion Planning Avoiding Dynamic Obstacles 103
4.1 Introduction . . . 103
4.1.1 Related Work . . . 109
4.1.2 Contribution . . . 112
4.1.3 Organization . . . 113
4.2 Preliminaries . . . 113
4.2.1 Linear Temporal Logic . . . 114
4.2.2 LTL Encoding for Multi-Robot Tasks . . . 116
4.2.3 Robot Dynamics . . . 119
4.3 Problem Formulation . . . 120
4.4 Approach . . . 124
4.4.1 Off-Line . . . 124
4.4.2 On-Line . . . 126
4.4.3 Integration of mission and motion planning . . . 127
4.5 Off-Line Synthesis: Resolving Deadlock . . . 128
4.5.1 Deadlock Resolution . . . 129
4.5.2 Resolving Deadlock when $m = 0$ . . . 132
4.5.3 Resolving Deadlock when $m = 1$ . . . 132
4.5.4 Resolving Deadlock when $m > 1$ . . . 134
4.6 Off-Line Synthesis: Environment Assumptions and Coordination 136
4.6.1 Runtime Certificates for the Environment . . . 137
4.6.2 Coordination Between Robots . . . 141
4.7 On-line Local Motion Planning . . . 142
4.7.1 Local Motion Planning Overview . . . 143
4.7.2 Constraints . . . 145
4.7.3 Optimization . . . 148

4.7.4 Deadlock Detection 149
4.8 Theoretical Guarantees 149
4.8.1 Correctness With Respect to Robot Dynamics 150
4.8.2 Collision-Free Motion 150
4.8.3 Correctness with Respect to the Task Specification 151
4.8.4 Computational Complexity 152
4.9 Experiments and Simulations 153
4.9.1 Synthesis and Revisions 155
4.9.2 Scalability with Respect to Dynamic Obstacles 156
4.9.3 Performance Evaluation 157
4.9.4 3D Problem Domain 161
4.9.5 Physical Experiments 161
4.10 Conclusion 163

Bibliography **165**

LIST OF TABLES

1.1 Comparison of several correct-by-construction control schemes. The techniques developed in Chapters 2 and 3 belong to the category indicated in boldface. 5

2.1 Fraction of (x_r,y_r,θ) coverage for each X_i associated with the FSM. 42

LIST OF FIGURES

1.1 A robot executing a user-assisted warehouse supply task. 2

2.1 Workspace and FSM for Example 2.1. In (a), a 2-D environment is shown, along with a set of trajectories: one that does not satisfy the controller (solid) and one that does (dashed). The •'s indicate when *S_blocked* turns from `False` to `True`. In (b), the number designates the state; the name in parenthesis denotes the region associated with that state. For each transition, the truth values for the *S_blocked* sensor are given; unlabeled transitions imply that *S_blocked* can take on any value. 13

2.2 Illustration of reactive composition. (a) shows a reach tube from R1 to R2 and possible trajectories. (b) shows a reach tube from R1 to R3 . Along any given trajectory leading to R2 (blue) in $\mathcal{L}_{12} \cap \mathcal{L}_{13}$, there exist $\mathcal{L}_{13}$ trajectories (red dashed) leading to R3 also in $\mathcal{L}_{12} \cap \mathcal{L}_{13}$. 20

2.3 Illustration of the reach tube computation steps, assuming symmetric transitions between each adjoining region. In (a), a pair of transition reach tubes $\mathcal{L}_{12}$ and $\mathcal{L}_{13}$ are computed for q_1 , the intersection of which (yellow) defines the new start set for the next iteration (see lines 8–14 in Algorithm 2.1). In (b), the same is done for the remaining states q_2 and q_3. Next, in (c), inward reach tubes $\mathcal{L}_i^c$ (red) are generated for each region, (see lines 15–18 in Algorithm 2.1). This expanded region defines the invariant for the next iteration. In (d)–(f), the process in lines 8–18 is again repeated for the new start sets and invariants, and terminates at (f) since all reach tubes lie inside the regions bounded by the dotted borders (e.g. for q_1 this is $R(q_1) \cap ((\mathcal{L}_{12} \cap \mathcal{L}_{13}) \cup \mathcal{L}_1^c)$). 26

2.4 (a) Workspace for the "patrolling two regions" example. (b) FSM for the example. The transitions are labeled with the truth value of the *pursuer* sensor; unlabeled transitions imply that *pursuer* can take on any value. 40

2.5 (a) shows a transition funnel for the transition from r_2 to r_3 and a slice of the set where r_1 is reachable from r_2 at $\theta = 1.35$ (defined by $\mathcal{L}_2^c \cup \mathcal{L}_{21}$) after the *first* iteration of Algorithm 2.1. (b) shows a 2-D view of the slice. In this iteration, the funnel is not reactively composable because the portion in r_2 is not confined to the red set. If the robot is outside of the red set when enroute to r_3, there are no controllers which can deliver it to r_1 if it sees a pursuer. . . 41

2.6 (a) shows a transition funnel for the transition from r_2 to r_3 and a slice of the set where r_1 is reachable from r_2 at $\theta = 1.05$ (defined by $\mathcal{L}_2^c \cup \mathcal{L}_{21}$) after the *second* iteration of Algorithm 2.1. (b) shows a 2-D view of the slice. The funnel is now reactively composable because the portion of it which lies in r_2 is now completely enclosed by the red set. The robot is therefore able to move to r_1 if it senses a pursuer anywhere along its path to r_3. 42

2.7 Closed-loop trajectory generated from an initial state in r_1. (a) shows a set of control laws are applied to implement the transitions (r_1, r_2) and (r_2, r_3), driving the robot from state 1 to state 2, then from state 2 to state 3 in Figure 2.4(b). 2-D projections of the active funnels are also shown, the red corresponds to inward and green corresponds to transition. *pursuer* turns `True` when at the location marked by the "+" sign. At this instant, another controller is invoked to make the transition (r_2, r_1). *pursuer* turns `False` at the "×" location, and new control laws are used to resume the transition (r_2, r_3). (b) shows an long-term path when the robot starts at the red "□" in r_1 and *pursuer* remains `False` throughout. 43

2.8 FSM for the pursuit-evasion example. For simplicity, we label each edge by the value the sensor proposition must take, and exclude labels for which the remaining input labels can be inferred based on the assumptions on the enemy's behavior. For example, the transition (q_6, q_7) is labeled *inR*2; by mutual exclusion of the regions the enemy can occupy, when *inR*2 is `True`, *inR*1 and *inR*3 are `False`. 44

2.9 Transition funnels for $\mathcal{L}_{56}$ and $\mathcal{L}_{67}$ (green) in the pursuit-evasion example. $\mathcal{L}_5^c$ and $\mathcal{L}_7^c$ are shaded blue and $\mathcal{L}_6^c$ is shaded red. Two hypothetical trajectories are shown; the one exiting the top of goal is not reactively composable and hence always enters r_4 regardless of the environment when in r_3, while the one exiting the left face of goal is reactively composable allowing the robot to choose between entering r_1 or r_2. 46

2.10 Partial sample trajectories of the unicycle robot (solid), and its polynomial approximation (dashed) for the delivery scenario in Example 2.1 . The robot pose is shown at uniform time intervals, and the projection of the funnels appears as shaded fill areas. The red ∘ indicates the rising edge of the *S_blocked* sensor. Note the yellow circled portion of the true system trajectory, indicating a place where it momentarily leaves the funnel. 49

2.11 Partial sample trajectories of the non-holonomic car (solid), and its polynomial approximation (dashed) for the delivery scenario in Example 2.1 . The robot pose is shown at uniform time intervals, and the projection of the funnels appears as shaded fill areas. The red ∘ indicates the rising edge of the *S_blocked* sensor. Note the yellow circled portion of the true system trajectory, indicating a place where it momentarily leaves the funnel. 50

3.1 Factory resupply example scenario. 54
3.2 Overview of the procedure for finding runtime certificates and controller synthesis. 68
3.3 2-D example. (a) shows the workspace map and grid whose cells are labeled with the configuration variable. The white grid cells denote r_1, while the gray denote r_2. (b) shows the synthesized controller for φ. 72
3.4 (a) shows a partial counterstrategy for Example 3.3 leading to deadlock. (b) shows a corresponding robot trajectory leading to deadlock. The cells shaded yellow indicate configurations in which there are no sequence of commands that avoid reaching r_2 eventually. (c) shows the result of a synthesized controller where deadlock is removed, but where the strategy *expects* the environment to set *sen* to `False` once the robot enters cell $x5$. The numbering of the cells correspond to the state labels, omitting the "x". 80
3.5 Map showing configurations for which the revisions ψ_t^e and ψ_t^s from Example 3.3 apply; a counterstrategy execution trace, as explained in Section 3.4.2. The green part of the path denotes where *sen* = `False` and the red denotes where *sen* = `True`. 81
3.6 (a) Partial visualization of the set of winning positions WP_{env}, showing all assignments to the environment proposition *sen* that are in the set WP_{env} in the next step in the execution given the system is currently occupying positions in the red-shaded cells and activating the indicated actions. (b) Map showing regions associated with cut states from Example 3.4. 86
3.7 Deadlock-free counterstrategy for Example 3.4. 87
3.8 A screen capture of the certificate visualization tool. The user specifies the current region and action by clicking regions in the workspace map (highlighted orange). Based on this input, the LTL formulas representing the revision matching that selection are displayed in the info box, along with a certificate of the revisions, provided as text-based feedback. Any sensors that are disallowed as per the command statement are painted red in the upper-right list. 93

3.9 Controller for φ in Example 3.1. Edges are labeled with the disjunction of assignments in X that may be assumed for that transition. 98
3.10 Continuous trajectories for the nonlinear unicycle abstraction in a 5×5 workspace, where the robot is initialized at the lower-left corner of the workspace. Dots along the trajectory indicate the position of the robot when a new control command is received (a time step of 0.35 seconds). Color indicates the state of the environment (red: s1_occupied; blue: s2_occupied). (a) shows a trajectory when the s1_occupied sensor is activated. (b) shows a trajectory when the s2_occupied sensor is activated. 99
3.11 Problem set up for the box-transportation scenario. The top image shows the KUKA youBot performing the task of delivering a box to the appropriate region as determined by the sensor. The map is displayed at the bottom left. The new region as a result of the reachability-based re-partitioning, $R_{middle,L}$, is shown in pink. The robot's trajectory is shown in black at the bottom right, along with the reachable sets for the activated controllers, and a nominal trajectory (magenta). A runtime certificate is generated (indicated in Figure 3.8) that indicates that the sensor should not change go_to_left to go_to_right when the robot is in $R_{middle,L}$. For full details, the reader is referred to [25]. 102

4.1 Surveillance/cleaning scenario. Two robots are tasked with actively monitoring the rooms of a museum. The robots must avoid collisions with static and moving obstacles and resolve deadlocks in order to achieve their goals. 106
4.2 Example of two connected regions. 118
4.3 Schema local and reference trajectories for an aerial vehicle, generated from the reference velocity $\mathbf{u}$. The tracking error is limited by ε and the robot volume dilated by ε. 120

4.4 Examples of integrated mission and motion planning. The blue robot starts in the region Goal 1 (top) and is tasked to visit Goal 2 (bottom right) and return to Goal 1. The red robot is placed in the region Goal 2 and is tasked to visit Goal 1 and return. The shortest path for both robots, given by solving a specification φ is to go through the corridor on the right. In (a), an execution of a specification φ using a local planner that locally avoids the collision between both robots and succeeds in executing the mission. (b) employs the same specification as (a), but the workspace is shrunk, resulting in a deadlock at location $\star$. (c) shows an execution of a controller synthesized from the modified specification φ'' using the deadlock resolution strategy and local planner developed in this work. With our approach, dynamic obstacles can be avoided locally, as in (a), and deadlocks can also be resolved. 121

4.5 Structure of the proposed mission and motion planner, with off-line and on-line parts. The mission planning is off-line and is described in Section 4.5 and in Section 4.6. The motion planner, Section 4.7, is computed at runtime and utilizes the strategy automaton (finite-state machine) synthesized off-line by the mission planner. 125

4.6 Diagram illustrating the deadlock resolution strategy for a single robot tasked with visiting $R1$ and $R8$. Starting in region $R1$ (marked '1'), the robot encounters deadlock (2) in region $R6$, while heading to $R7$. The $R6$-to-$R7$ transition is prevented (red line), and the robot must move a discrete radius m away from the deadlock event to resolve deadlock. If $m = 1$, then deadlock is resolved once the robot crosses the green line, leaving $R6$ (3a). From there, it may reach $R8$ (4a) if no other deadlocks are encountered. On the other hand, when $m = 3$, deadlock is resolved only when crossing the cyan line (3b); an alternate path to the goal may result (4b). 129

4.7 Schema local and reference trajectories for an aerial vehicle, generated from the reference velocity $\mathbf{u}$. The tracking error is limited by ε and the robot volume dilated by ε. 145

4.8 Workspace showing specification revisions for each region completion/activation pairs where singleton or pairwise deadlock may occur. An arrow's color indicates the type of assumption that has been made. The number (or pair of numbers) indicates the robot (or robot pairs) concerned with the assumption. The placement of the arrow indicates the region that the robot is headed (i.e. its action commands $AP_{\mathcal{R}}^{act}$) when the given assumption holds true. 156

4.9 Example of the approach in a scenario with six dynamic obstacles (dark red) and one controlled quadrotor (light green/yellow). The path of the controlled quadrotor is shown with a dashed green line. (a) The original approach without deadlock resolution avoids collisions but can get into unresolved deadlocks. The path leading to the deadlock is shown. (b) The proposed approach successfully resolves deadlocks, like the one shown here. The path leading to and resolving the deadlock are shown. (c) Path of the controlled quadrotor using the proposed approach during a ten minutes simulation. The quadrotor successfully avoids collisions and reverts the motion when it encounters a deadlock. 158
4.10 Example of the approach with six dynamic obstacles (dark red) and two controlled quadrotors (light green). The dynamic obstacles navigate to randomized locations and the controlled robots execute the proposed framework. The path of the controlled quadrotors is shown with a dashed green line for one minute of the simulation. The quadrotors successfully avoid collisions, reverse motion when they encounter a deadlock and explore the top and bottom rooms. 159
4.11 Comparison of the results of the "garbage collection" scenario with DOs exhibiting counter-flow (CF) or random waypoints (RW) behaviors, with either one quadrotor (1Q) or two quadrotors (2Q). For each scenario, we evaluate the results for data collected over 40 200-sec runs. Six quadrotors were used for the DOs. Over each run, (a), (b), and (c) show, respectively, data for the number of encountered deadlocks, the number of goals visited, and the number of unresolvable deadlocks. Standard deviations are indicated as error bars in (a) and (b). 160
4.12 Deadlock resolution (green robot) and safe navigation in a 3D environment. Quadrotors are displayed at the final time and their paths for the time interval. Each yellow disk represents a quadrotor and the cylinder its safety volume. The orange robot represents the dynamic obstacle. 162
4.13 Planar scenario with two centrally-controlled Nao robots and a dynamic obstacle (youBot). In each image, three consecutive frames of the robot's motion are superimposed. In (a), the local planner enables the two Naos to avoid collisions with each other. In (b), one of the Naos reverses direction to resolve the deadlock with the youBot. 163

CHAPTER 1

INTRODUCTION

With each robot that arrives on the market, new ideas are sparked for tasks we wish for them to accomplish. For instance, programming the Atlas[1] to accomplish a task such as package sorting and delivery would be cumbersome to carry out manually. Over the last decade, automated, correct-by-construction synthesis frameworks have been developed for robotics applications to guarantee tasks in unpredictable environments [10, 38, 42, 46, 52, 61, 91]. The common feature among such approaches is the automatic synthesis of controllers from a formal, high-level specification and geometrical description of the environment. This essentially removes the need to program, by hand, each action for the robot to take and then exhaustively check that controllers that use these actions actually satisfy the specification under all possible conditions faced at runtime. Synthesis approaches that are able to reason about a dynamically-changing environment are dubbed *reactive synthesis*; such systems enable sensor-rich systems to execute tasks in dynamic, human environments in a provably-correct manner.

Reactive synthesis addresses the problem of finding a controller that guarantees tasks expressed in a formal specification language, while reacting to the presence on an environment that evolves over time. We adopt linear temporal logic (LTL), a formal specification language that enables users to specify the required guarantees that the system must uphold over an infinite horizon, as well as any assumptions placed on the dynamic environment. For instance, given the robot and workspace shown in Figure 1.1, a user may write the specifica-

[1] an anthropomorphic robot with several degrees of freedom per limb

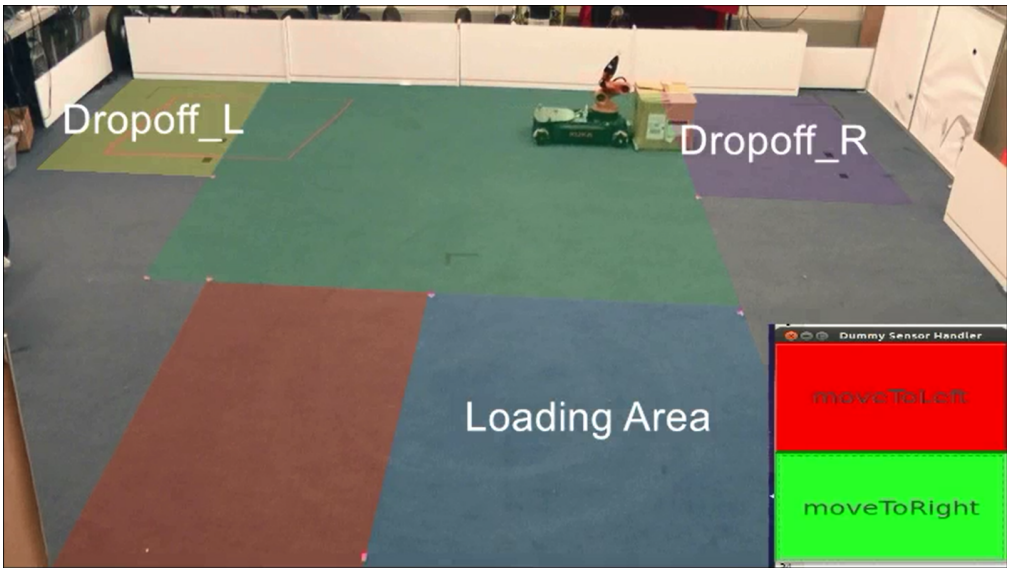


Figure 1.1: A robot executing a user-assisted warehouse supply task.

tion: "If you see a box, move to the pick-up location. If you are carrying a box, then visit a requested drop-off location. Always avoid people."

While many approaches focus on simple robot models, in general, the existence of a satisfying controller relies on the the ability to extend high-level guarantees (on a discrete abstraction of the system) to a potentially complex, nonlinear dynamical system. While recent activity toward this end has focused on frameworks that use a combination of on-line and off-line approaches for correct-by-construction controller synthesis for nonlinear systems (e.g. [10, 20, 59, 64, 89, 91]), the overall goal of this work is an approach that will provide, *at synthesis time*, the guarantee that the task is implementable on a given physical system.

The first contribution is an approach for synthesizing controllers that adhere to the reactive composition properties for the given task [20]. To do so, we introduce a new composition property to assure that the controllers produce

continuous trajectories which guarantee the reactive behaviors of a finite-state machine (FSM) synthesized for the task. The algorithm takes as an input a high-level controller with reactive behaviors, and tries to automatically synthesize a library of low-level (atomic) controllers that are guaranteed to satisfy these behaviors. To the authors' knowledge, ours is the first attempt at synthesis of verified controllers for reactive tasks using nonlinear systems.

The second main contribution addresses cases where there exists no satisfying controller for the mission, given the dynamical system and the environment's assumed behaviors [21]. From an LTL mission specification, we introduce an approach that leverages information about an abstraction of the dynamical system to automatically generate a concise set of revisions to such specifications. We provide a graphical visualization tool as a design aid, allowing the revisions to be conveyed to the user interactively and added to the specification at the user's discretion. Any accepted statements become certificates that, if satisfied at runtime, provide guarantees for the current mission on the given dynamics. Our approach is cast into a general framework that works with various discrete representations (i.e. abstractions) of the system dynamics.

The final contribution is a method for automatically synthesizing reactive controllers for a team of robots with non-trivial dynamics operating in a workspace shared with other agents, such as humans or other robots [2]. From a mission specification written in LTL, we synthesize an automaton representing a strategy for a robot's motion that guarantees the specification. This automaton is executed at runtime in conjunction with a distributed local motion planner that guarantees avoidance of collisions with dynamic obstacles. Our contribution is a correct-by-construction synthesis approach to multi-robot mission

planning that guarantees collision avoidance and resolves encountered deadlocks. The proposed method provides environment assumptions under which deadlock will be avoided by identifying environment behaviors that, when encountered at runtime, may prevent the robot team from achieving its goals. The abstraction approach yields high-level controllers that provide guarantees even when a high density of dynamic obstacles is present in the environment. As such, the proposed synthesis approach overcomes the hurdle of scalability in multi-agent mission planning. We demonstrate the off-line synthesis approach and on-line execution of the controllers with aerial vehicles (quadrotors) navigating in 2D and 3D environments and with walking humanoids moving in 2D environments.

1.1 Related Work Overview

In Table 1.1, some key technical features of the frameworks proposed in Chapters 2 and Chapter 3 are compared with similar correct-by-construction approaches reported in the literature. The reader is referred to Chapter 4 for a similar comparison of related work in multi-agent controller synthesis.

Approaches are categorized according to three factors: whether or not they are suited to controlling nonlinear systems, whether they express tasks that are reactive to a dynamic environment or assume a static environment, and whether the algorithms compute the controller on-line (at runtime) or off-line. If the approach is capable of handling nonlinear systems, the specific classes of systems reported by the author are provided. The contribution of this thesis is on addressing reactive specifications and nonlinear systems; therefore, the remainder

Table 1.1: Comparison of several correct-by-construction control schemes. The techniques developed in Chapters 2 and 3 belong to the category indicated in boldface.

Handles linear systems		Handles nonlinear systems	
Reactive tasks, off-line synthesis	[36, 52]	**Reactive tasks, off-line synthesis**	[59][2], ([22, 25], Ch. 2)[3], ([57], Ch. 3)[4]
Reactive tasks, on-line computation	[91]	Reactive tasks, on-line computation	[93][5], [10, 64][6]
Non-reactive tasks, off-line synthesis	[6, 46, 94]	Non-reactive tasks, off-line synthesis	[89][7], [45][8], [42][9]

of this section compares, in greater detail, the works that are most similar to ours - nonlinear controller synthesis for reactive tasks (the two entries in the upper-right corner of Table 1.1).

The funnels-based controller synthesis framework discussed in Chapter 2, in particular, handles nonlinear systems whose dynamics are polynomial in the states and inputs. The approach furthermore handles reactive task specifications in an off-line synthesis approach where the strategy for the robot to follow is determined before runtime. The work bears similarity to approaches such as [59]; however, the distinction to be made is the dynamics used in [59] are restricted to the class of differentially-flat systems, which applies to many in-

[2]Nonlinear differentially flat systems
[3]Nonlinear polynomial systems and nonlinear hybrid polynomial systems
[4]Discrete-state abstractions of nonlinear systems on uniform grids
[5]Discrete-state abstractions of nonlinear time-invariant systems
[6]General smooth nonlinear systems
[7]State-constrained nonlinear systems
[8]Nonlinear hybrid systems
[9]General smooth nonlinear systems

teresting robotic systems (wheeled robots and simple car models) but leave out others (aircraft, 3D manipulators). Our work, on the other hand, provides an approach to capture the class of polynomial systems inclusive of many robots, such as mobile robots and manipulators, but may require approximations when more complex dynamics must be assumed.

Other off-line controller synthesis approaches assume even wider classes of nonlinear systems, provided that the systems can be represented using *discrete abstractions* - a discrete approximation of the system usually over a gridding of the state space. The runtime certificates approach presented in Chapter 3 is one such case, where an abstraction-based approach similar to that reported in [57] is used. Where the proposed synthesis approaches differs from [57] is in the ability to handle the assumptions on the environment's behavior; our approach adds a feedback channel to alert the user of the certificates generated during the synthesis process. The approach of [57], in contrast, does not offer user feedback but is suited for the case when the high-level controller must be robust to bounded uncertainties; for instance, if sensor noise is experienced at runtime.

There are a few key differences between control approaches that are computed at runtime, such as on-line control approaches of [10, 64, 93], and those whose mission plans are completely determined off-line before the plans are executed. Except under certain assumptions on the system model, environment, or specification, on-line control schemes generally do not have the ability to guarantee that the resulting controller will be able to achieve its specified goals safely. In [93], for instance, a receding-horizon controller defines control actions taken over a finite horizon at a particular sampling instant, and the controller is

re-computed at every subsequent instant based on new observations captured at each step. There is a possibility that the environment may not behave in a way that leads to satisfaction of the goals, for instance, if a door shuts behind the robot. In our work, the set of states that guarantee the entire task are computed at synthesis time, and the plan remains fixed when applied at runtime. Since off-line approaches generally require the computation and storage of a complete mission plan and controllers for use at runtime, there are usually greater demands on storage than their on-line counterparts. Also, in dealing with a dynamic environment, the user is required to state all of the assumptions on the environment as part of the specification. While this can be cumbersome for a user, the objective of Chapter 3 is to alleviate this burden somewhat by formulating a synthesis paradigm that provides revisions to the user without requiring him/her to specify every assumption by hand.

CHAPTER 2

SYNTHESIS OF NONLINEAR CONTINUOUS CONTROLLERS FOR VERIFIABLY-CORRECT HIGH-LEVEL, REACTIVE BEHAVIORS

2.1 Introduction

As robots become more sophisticated, we see increasing potential for them to perform complex tasks. Sensor-rich platforms such as self-driving cars, robotic manipulators, humanoids, and unmanned air vehicles enable capabilities ranging from human-assistance to search-and-rescue to exploration. To operate effectively in the real world, a robot must be able to perform tasks in a way that avoids causing harm to itself or the people it encounters, by appropriately *reacting* to the environment. Tasks that are *reactive* require that the robot's behaviors change in response to real-time sensory information. It is therefore imperative that the controllers we provide to these robots produce behaviors that are guaranteed to satisfy all task instructions provided to the robot given our knowledge of the environment. Building upon existing concepts from the controller verification and motion planning communities, we present an automated methodology for computing (if possible) a set of verified controllers that fulfill the reactive behaviors of a high-level task.

In the controller synthesis literature, recent activity has focused on solving the "reach-avoid" problem for nonlinear systems. Methods based on a game-theoretic representation of the problem have been solved [28], as have game-theoretic solutions involving a sequence of goals [80]. Rather than working directly with the nonlinear system, other researchers compute symbolic abstractions of the underlying system model [95] for which efficient algorithms exist

for the exact computation of reachable sets. The particular formulation in [95] enables the synthesis of hybrid controllers for goal satisfaction with static obstacles [65].

Concepts from nonlinear controller verification have been adopted in the motion planning domain. For example, libraries of verified motion plans have been generated for hybrid systems [41] as well as for nonlinear polynomial systems [82]. Both use a sampling-based strategy and employ solution methods that are efficient enough to allow on-line re-planning based on runtime sensor information [63]. The latter method ([63, 82]) builds upon the controller sequencing work of [13] by combining sampling-based motion planning with local feedback controllers to define a robust neighborhood (funnels) computed about one of the sample trajectories. Controller sequencing has been applied to specific robotic applications; for example, in [17], local verified motion controllers have been devised for robots with nonholonomic kinematic models. While each of these methods provide the framework for motion planning in the context of safety and reachability, they are not immediately equipped to handle automatic synthesis of controllers for *reactive* tasks with possibly infinite duration.

Ongoing work in the robotics community has been devoted to developing provably correct synthesis algorithms for complex robot task specifications. A variety of techniques exist for synthesizing controllers for non-reactive tasks [10,38,42,46,61], as well as those for reactive tasks [52,93]. The typical workflow is a three-step process: abstract the system model and the robot's workspace; perform synthesis based on the abstraction and task specification; and design low-level controllers fulfilling each of the behaviors in the high-level controller.

Low-level controller design can be performed using potential functions [18], vector fields [7], or rapidly-exploring random trees (RRTs) [54], to name a few. While such strategies often suffice for fully-actuated robots, correctness guarantees usually do not extend to robots with more complicated dynamics. The work of [50] describes an application of provably-correct reactive synthesis to cars with a rear-drive Ackermann steering model. In that work, the authors manually designed a palette of controllers specific to the robot model that could be automatically instantiated in different environments. Here we describe algorithms for automatically generating such controllers for a wider class of nonlinear robot models. Some researchers have extended provably correct controller synthesis to nonlinear systems [59, 89]. These methods, however, rely on finding suitable discrete abstractions for the nonlinear system model, if any such abstractions exist. Others have introduced a multi-layered synthesis strategy in which the high-level controller is designed off-line, while the motion planning layer processes the high-level requests by accounting for nonlinear dynamics and any encountered workspace obstacles [10, 64]. However, the tasks that are considered in those works are non-reactive and the task fulfillment guarantees are obtained at runtime, rather than at the time of synthesis. Synthesis of provably-correct controllers for a complex vehicle in the presence of uncertainty is treated in [16]. The approach takes into account uncertainty bounds on the evolution of trajectories over time, however a challenge is in the application to reactive task specifications under indefinite execution durations. In the work of [35], an automated framework is introduced for translating the behaviors of a high-level controller into low-level continuous controller specifications. The possibility exists to use such specifications in the design of nonlinear motion planners.

2.1.1 Problem Statement

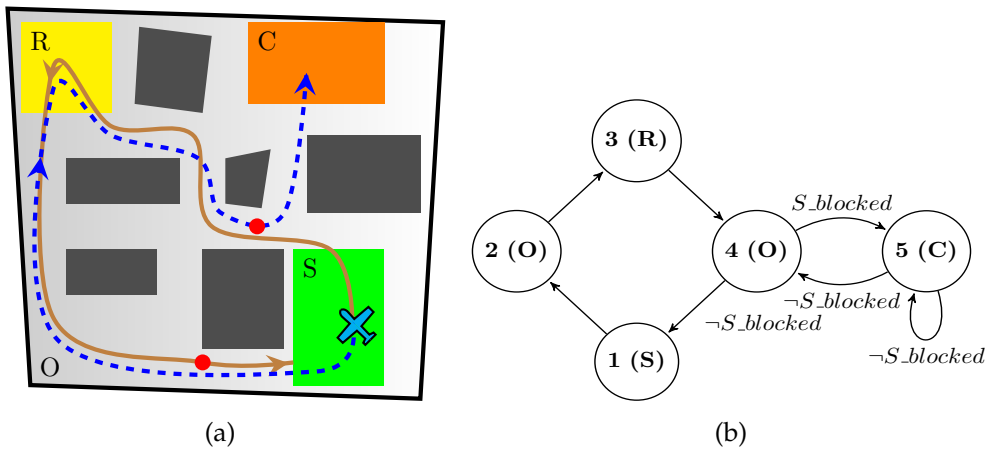
In this chapter, we address the following two questions. Given a high-level reactive mission plan (a control strategy consisting of deterministic region transitions that satisfy a user-defined *mission specification*), a robot model, and a workspace, can we design a set of low-level controllers that guarantee satisfaction of the task in a dynamic environment? If so, what is the set of allowable robot configurations for which these guarantees hold?

To be precise, we are given a deterministic finite-state machine (FSM), the nonlinear differential equations of the robot, and a representation of the robot's workspace. The FSM represents a control strategy synthesized from a high-level mission specification using, e.g., the approach described in [11]. The remaining problem - the focus of this chapter - is devising a synthesis method that automatically generates a collection of low-level atomic controllers (if any exist) to realize each of the transitions in the FSM. In contrast to receding-horizon controllers (e.g. [85, 93]) that are computationally expensive to implement on resource-constrained robots, the approach we adopt is based on a set of switched state-feedback controllers that are all precomputed offline. Thus, we aim for an efficient implementation at the expense of a slightly larger offline computational load compared to existing approaches. We work directly with coarsely-partitioned workspaces rather than synthesize controllers on a grid as in [58]. This is to alleviate the approximations needed to generate discrete abstractions and the complexity issues as workspaces are scaled larger.

The main contributions of this work are as follows. First, building from the notion of sequential composition introduced in [13,69], we define a new composition property, which we call *reactive composition*, to assure that the derived con-

trollers produce continuous trajectories which guarantee the reactive behaviors of the FSM. That is, if the environment changes part-way through a transition, the robot must be able to correctly switch to a different transition. We do this by creating a set of constraints for the construction of low-level controllers. This part of the work is inspired by the general approach in [35], however, in our work we extend controller specifications to encompass nonlinear robot models satisfying high-level reactive tasks. Another contribution of our work is an algorithm for synthesizing controllers that adhere to the reactive composition properties for the given task. The algorithm takes as an input a high-level controller with reactive behaviors, and tries to automatically synthesize a library of low-level (atomic) controllers that are guaranteed to satisfy these behaviors. To the authors' knowledge, ours is the first attempt at designing verified controllers for reactive tasks. Finally, we contribute a sample-based method for computing controllers that implement each specific transition in the high-level controller, avoiding any unintended behaviors. We adapt the invariant funnels method in [62, 82], which takes advantage of powerful numerical optimization techniques to produce a set of verifiably-safe atomic controllers. We introduce a new set of constraints in order to enforce reactive composition while preventing any robot behaviors prohibited by the high-level controller.

In the authors' preliminary version of this work, [24], we introduced the concept of reactive composition, developed a constructive algorithm to ensure this property, and presented preliminary simulation results. This work serves to build upon that work by detailing the precise technical conditions upon which reactively composable reachable sets should be constructed. Also, we explain the numerical method we use to compute atomic controllers, in particular our approach for incorporating the necessary invariance conditions for the reactive



(a) (b)

Figure 2.1: Workspace and FSM for Example 2.1. In (a), a 2-D environment is shown, along with a set of trajectories: one that does not satisfy the controller (solid) and one that does (dashed). The •'s indicate when *S_blocked* turns from `False` to `True`. In (b), the number designates the state; the name in parenthesis denotes the region associated with that state. For each transition, the truth values for the *S_blocked* sensor are given; unlabeled transitions imply that *S_blocked* can take on any value.

setting. We furthermore provide more illustrative examples in more complex environment settings.

We motivate the work through an example.

Example 2.1. *Consider an autonomous delivery robot, modeled as fixed-wing aircraft, operating in a dense urban environment in Figure 2.1(a). The robot must continually deliver items between the store (S) and the restaurant (R) by visiting them infinitely often. If the plane is in O and en route to S but it senses that store is blocked, then it must re-route to the charging station C (without entering S). We adopt the synthesis approach in [11] to construct the FSM shown in Figure 2.1(b).*

Our goal is to construct feedback controllers that guarantee the sequence of motions of the FSM like the one in Figure 2.1(b) using, in this case, a fixed wing airplane model and the workspace in Figure 2.1(a). This task is a reactive one be-

cause the behavior changes depending on whether or not the store is "blocked." If the robot starts in S and $S\_blocked$ stays `False` forever, the robot should follow the sequence of regions $SOROS\ldots$ indefinitely; if $S\_blocked$ becomes `True`, then it may follow the sequence $SOROCC\ldots$, i.e. the robot must go to C and stay there once it has visited R. In the continuous domain, there may be instances where the robot may fail this task. To see this, consider the solid-line trajectory pictured in Figure 2.1(a), where the plane moves counterclockwise starting with $S\_blocked$ = `False`. If the robot senses $S\_blocked$ = `True` at the •, it may be unable to avoid hitting S, and the task would fail. On the other hand, a clockwise trajectory (the dotted line in Figure 2.1(a)), is likely to succeed in this workspace. In general, it is possible that $S\_blocked$ may toggle between `True` and `False` at any point in the robot's continuous trajectory, and so the robot must always be in a configuration where it can make any legal transition. If, for a different workspace, such controllers are found not to exist for the given platform, then the specified task may may not be suitable for that robot.

2.1.2 Chapter Outline

In the next section, we introduce the reader to relevant background concepts for our atomic controller design approach. In Section 2.3, we introduce our approach to solving the problem and outline an algorithm for automatic controller synthesis. The trajectory-based approach used for computing verified controllers is presented in Section 2.4, and the execution of the controllers in Section 2.5. We then present simulations of tasks performed by different robots in Section 2.6. The chapter concludes in Section 2.7 with a summary and discussion.

2.2 Preliminaries

2.2.1 Robot and Environment Abstractions

Similar to the implementations in [52], we define an abstraction as a partitioning of the robot system and the environment in which it is operating. In particular, we partition the continuous map $\mathcal{M} \subset \mathbb{R}^n_w$ into M disjoint proposition-preserving map regions $\mathcal{M}_i$, each associated with a label r_i, where n_w is the dimension of the workspace. In a planar environment, for example, the workspace is made up of two-dimensional polygons, some of which may represent holes in the map. We also assume that the robot can travel between any adjacent regions.

In addition to the workspace, sensor and action values are discretized such that they may be effectively replaced by a set of Boolean propositions. Discretization is achieved by dividing the continuous space into a finite number of equivalence classes and assigning unique propositions to each class [52]. In this work, we assume that sensors and actions are binary. Here sensors refer to events external to the robot (detection of a person, non-detection of a person), whereas actions refer to discrete robot functions (pick up, drop). We define $\mathcal{X}$ as a set of *environment propositions*, collecting sensor propositions, and $\mathcal{Y}$ as a set of *system propositions*, collecting robot action propositions and region propositions. Define the set $\mathcal{R} = \bigcup_i r_i \subseteq \mathcal{Y}$ as the subset of $\mathcal{Y}$ corresponding to regions.

2.2.2 Controller Finite-State Machine

High-level controllers synthesized from task specifications and abstractions [52, 93] are assumed to be given in this work. The synthesized controller takes the form of a FSM A, defined as a tuple $A = (\mathcal{X}, \mathcal{Y}, Q, Q_0, \delta)$, where:

- $\mathcal{X}$ and $\mathcal{Y}$ are proposition sets as defined in Section 2.2.1.
- $Q \subset \mathbb{N}$ is a set of discrete states.
- $Q_0 \subseteq Q$ is a set of initial states.
- $\delta : Q \times 2^{\mathcal{X}} \to Q$ is a deterministic transition relation, mapping states and subset of environment propositions to successor states.

We introduce the following additional definitions. Define $\gamma_{\mathcal{R}} : Q \to \mathcal{R}$ as a state labeling function assigning to each state the region label for that state, r_i. Define the operator $R : Q \to \mathbb{R}^n$ as a mapping that associates with each $q \in Q$ the subset $X_q = R(q)$ of the free configuration space X, where X_q corresponds to an n-D polytope labeled with $\gamma_{\mathcal{R}}(q)$. For example, a nonholonomic planar mobile robot with $X \subset SE(2)$ would use 3-D polytopes for X_q; X_q for higher-dimensional systems would be polytopes of appropriate dimension. We also define Δ to be the collection of state-pairs corresponding to each transition, namely $\Delta = \{(q, q') \in Q^2 \mid \exists z \in 2^{\mathcal{X}} \,.\, \delta(q, z) = q'\}$. Furthermore, we assume that actions other than locomotion are instantaneous.

For some sequence $w(0)w(1)w(2)\ldots$, $w(i) \in 2^{\mathcal{X}}$, $i = 0, 1, 2, \ldots$, denote a *run* of A as $q(0)q(1)q(2)\ldots$, with $q(i+1) = \delta(q(i), w(i))$. We call the sequence $w(0)\gamma_{\mathcal{R}}(q(0))w(1)\gamma_{\mathcal{R}}(q(1))\ldots$ an *execution trace of A*.

2.2.3 Continuous Dynamics and Operations

We consider systems of the form

$$\dot{x} = f(x, u, d), \quad x(0) \in S, \quad d \in D, \quad u \in U \tag{2.1}$$

where we are given a state vector $x \in X \subseteq \mathbb{R}^n$, a control vector $u \in U \subset \mathbb{R}^m$, a vector of unknown disturbances $d \in D \subseteq \mathbb{R}^{n_d}$, and a set of initial states S. f is considered to be a smooth, continuous vector field with respect to its arguments.

Under the action of a full-state feedback control law $u(t) = \kappa(x(t), t)$, let $\dot{x} = \hat{f}(x, d)$ represent the closed-loop system for (2.1) under $\kappa(x(t), t)$. Given a concrete start state $x(0) \in S$ and a time horizon $T > 0$ (a free parameter that will be discussed in Section 2.3.1), denote $\xi_T : [0, T] \to X$ as a continuous finite-time trajectory of states under $\hat{f}$, $\mu_T : [0, T] \to U$ as a trajectory of control inputs, and $\xi_T^d : [0, T] \to D$ as a trajectory of disturbances. Say we are given a sequence of time indices, $\mathbf{t} \in \{0, \ldots, T\}$; then, we can represent the continuous trajectory as a sequence of states $\mathbf{x} = \{\xi(\tau)\}_{\tau \in \mathbf{t}}$, and a sequence of control inputs $\mathbf{u} = \{\mu(\tau)\}_{\tau \in \mathbf{t}}$.

Given a smooth function $V(x, t)$, some $\rho(t) > 0$, and some T, we define the *ρ-sub-level set* as

$$\ell(\rho(t), t) = \{x \mid V(x, t) \leq \rho(t)\}, \qquad \forall t \in [0, T].$$

Let $X_i, X_j \subset X$ be regions in the configuration space where $X_i \cap X_j \neq \emptyset$ (the regions are adjacent). Define an *atomic controller* κ_{ij} as a controller that steers $f(x, u, d)$ from X_i to X_j without leaving $X_i \cup X_j$ under all possible disturbances. Formally, an atomic controller κ_{ij} exists iff there exists some initial state $\xi_T(0) \in X_i$ and some relative final time T_{ij} such that $\xi_T(T_{ij}) \in X_j$ and $\xi_T(t) \in X_i \cup X_j$ for all $\xi_T^d(t)$, $t \in [0, T_{ij}]$. We leave T_{ij} as a free parameter to be chosen by the algorithm, as

will be discussed subsequently. We refer to trajectories driven by the action of an atomic controller as *atomic trajectories*.

Lastly, given $X_i, X_j \subset X$, define a *reach tube* $\mathcal{L}_{ij} : [0, T_{ij}] \to X_i \cup X_j$ as the set of trajectories in which the controlled system remains for $t \in [0, T_{ij}]$ under the action of a feedback controller. Formally, suppose that we define some initial set S_{ij} and some atomic control law κ_{ij}, then $\mathcal{L}_{ij} = \{\xi_T(t) \mid \xi_T(0) \in S_{ij}, \forall \xi_T^d(t), t \in [0, T_{ij}]\}$. We write $\mathcal{L}_{ij}(t)$ to denote a slice of the reach tube $\mathcal{L}_{ij}$ evaluated at time t. In the following, we abuse notation and write, for example, $X_i \cup \mathcal{L}_{ij}$ to express the union of X_i and the set covered by the reach tube (in place of $X_i \bigcup_k \mathcal{L}_{ij}(t_k)$).

2.3 Controller Synthesis Approach

To lay the foundation for our approach, in this section we introduce different classes of atomic controllers and the technical conditions that allow us to guarantee behaviors in the high-level controller. Lastly, we describe an algorithm which takes as its input the FSM and returns a library of atomic controllers that guarantee the continuous executions of the FSM at runtime.

2.3.1 Composition Strategies

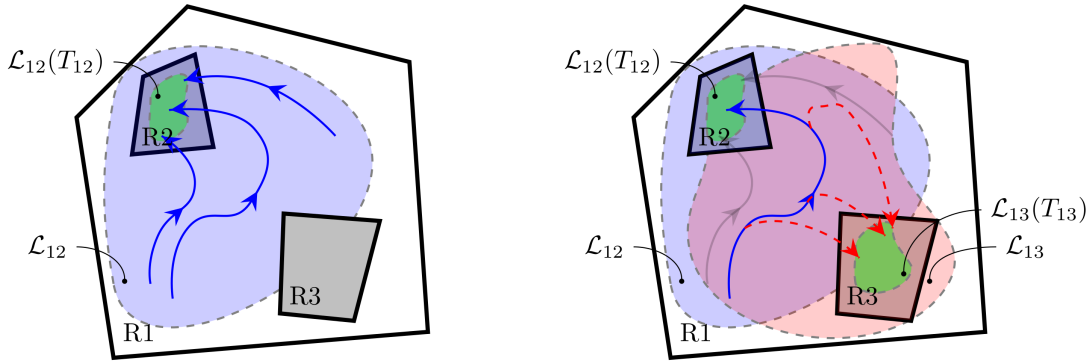
Define $I^i_{out} = \{k \in \mathbb{N} \mid (q_i, q_k) \in \Delta\}$ as the index set of all successor states for state q_i (e.g. for state 4 in Figure 2.1(b), $I^4_{out} = \{1, 5\}$), and let $I_{out} = \{I^i_{out}\}_{q_i \in Q}$. As defined by [13], for a given set of atomic controllers to be *sequentially composable*, we require goal sets of the reach tubes to be contained within the domain of successor reach tubes. Formally:

Definition 2.1 (Sequential Composition [13]). *Let $\mathcal{L}_{ij}(t)$ denote the slice of $\mathcal{L}_{ij}$ at time t. For $\mathcal{L}_{ij}$ to be sequential composable for each pair in Δ implies that $\mathcal{L}_{ij}(T_{ij}) \subseteq \mathcal{L}_{jk}$ for all $k \in I^j_{out}$.*

For reactive tasks, the underlying conditions for sequential composition do not capture the possibility that the robot might need to change its motion part-way during a trajectory in response to some event. Consider again Example 2.1 and the two trajectories pictured in Figure 2.1(a). When the robot is in O with $S\_blocked$ = `False`, and moving towards S, both trajectories may in fact satisfy the sequential composition property if atomic controllers exist for transitions between R and O and O and S. However, if $S\_blocked$ turns `True` during the execution, the robot must already be in a state where it may access C *without* first entering S (a violation of the specification). The configurations where S is reachable from O must also be the set of configurations where C is reachable from O. We introduce a constraint called *reactive composition* to deal with co-reachability of successor regions.

Definition 2.2 (Reactive Composition). *Let $\bar{X}_i \subset X$ denote the set of states such that, for all $q_i \in Q$, there exists a trajectory from q_i to any q_k, $k \in I^i_{out}$, i.e. $\bar{X}_i = \bigcap_{k \in I^i_{out}} \mathcal{L}_{ik}$. A given reach tube $\mathcal{L}_{ij}$ is* reactively composable *with respect to A if, for $(q_i, q_j) \in \Delta$, for all state trajectories $\xi_T \in \mathcal{L}_{ij} \Rightarrow \xi_T \in \bar{X}_i \cup \bar{X}_j$.*

Reactive composability is illustrated in the 2-D scenario in Figure 2.2(b), where the solid blue trajectory shown exiting region $R1$ (corresponding to state 1) and entering $R2$ (state 2) is both contained completely within its own reach tube (shaded blue) and the reach tube (shaded red) for trajectories leading to $R3$ (state 3). Note that T_{12} and T_{13} signify the times beyond which all trajectories in $\mathcal{L}_{12}$ and $\mathcal{L}_{13}$ have reached their respective goal regions. As discussed next, we



(a) (b)

Figure 2.2: Illustration of reactive composition. (a) shows a reach tube from R1 to R2 and possible trajectories. (b) shows a reach tube from R1 to R3 . Along any given trajectory leading to R2 (blue) in $\mathcal{L}_{12} \cap \mathcal{L}_{13}$, there exist $\mathcal{L}_{13}$ trajectories (red dashed) leading to R3 also in $\mathcal{L}_{12} \cap \mathcal{L}_{13}$.

fulfill this property by imposing additional constraints on the reach tubes we generate.

2.3.2 Classes of Atomic Controllers

We now introduce two types of atomic controllers for constructing reactively composable controllers, and describe the constraints we apply in each case. *Transition* controllers κ_{ij} refer to those which invoke a transition between adjacent regions. *Inward-facing* controllers κ_i^c are used to maximize coverage of the region. Respectively, the reach tubes for κ_{ij} and κ_i^c are $\mathcal{L}_{ij}$ and $\mathcal{L}_i^c$.

Transition Reach Tubes, $\mathcal{L}_{ij}$

Consider a pair $(q_i, q_j) \in \Delta$. When we construct reach tubes for this transition, we want it to satisfy the current transition in the context of all possible subsequent

transitions, that is, satisfy the reactive composition requirement. To this end, we introduce the following three conditions.

1. Trajectories must be *atomic* with respect to the reactively composable set $\bar{X}_i \cup \bar{X}_j$: we want trajectories to start in the set $R(q_i)$ and progress to $R(q_j)$ while remaining in an invariant Inv_{ij} (defined in Section 2.3.2). For now, we apply definition 2.2 by choosing $Inv_{ij} = \bar{X}_i \cup \bar{X}_j$.
2. Trajectories must reach a goal set $G_{ij} \subseteq X_j$; formally, $\xi_T(T_{ij}) \in G_{ij}$ after some time T_{ij} has elapsed relative to the start of the trajectory. We require G_{ij}, whose precise definition will become clear in Section 2.3.2, in order for the reach tube to be sequentially composable with reach tubes for successor states from q_j. For now, we will assume G_{ij} to be $\bar{X}_j$.
3. To prevent the robot from re-entering a region X_i once it has entered X_j for $X_i \neq X_j$, the trajectories need to be invariant in finite time to the set X_j. That is, for some $\tau \in [0, T_{ij}]$,

$$\xi_T(\tau) \in \partial X_j \Rightarrow \xi_T(t) \in X_j, \quad \tau < t \leq T_{ij}$$

   where we denote ∂X_j to mean the boundary of the set X_j.

To devise a certificate for the third condition, we can draw from region-of-attraction analysis [43], as follows. Let $V : \mathbb{R}^n \times [0, T_{ij}] \to \mathbb{R}$ be a smooth differentiable function with $V(\xi_T(t), t) = 0$ and $V(x, t) > 0, x \neq \xi_T(t)$. We further restrict $V(x, t)$ to be bounded from below and above by class-$\mathcal{K}$ functions for all $t \in \mathbb{R}_+$. [43][1] We therefore want to ensure that the level set $\partial\ell(\rho(t), t) = \{x \mid V(x, t) = \rho(t)\}$ satisfies $\dot{V}(x, t) = \frac{d}{dt}V(x, t) < 0$ on $\{x \mid x \in \partial\ell(\rho(t), t) \cap X_j\}$. Put in terms of the closed-loop system $\hat{f}_{ij}(\cdot, \cdot)$, the third condition can be reduced to the easier problem of

[1] In Section 2.4, we will be choosing $V(x, t)$ as a quadratic that satisfies these conditions.

restricting $\rho(t)$, the size of the attraction region, such that it satisfies:

$$\dot{V}(x,t) = \frac{\partial}{\partial x}V(x = \xi_T(t), t)\hat{f}_{ij}(\xi_T(t), d) + \frac{\partial}{\partial t}V(\xi_T(t), t) < 0, \quad \xi_T(t) \in \partial X_j \cup \ell(\rho(t), t) \neq \emptyset, \quad \forall d \in D.$$

Intuitively, this statement requires that the system flow toward the successor region only on that segment of the region boundary which is also in the computed region of attraction.

We can satisfy these conditions by simultaneously imposing constraints on the construction of $\mathcal{L}_{ij}(t)$. Respectively, these constraints are:

$$\mathcal{L}_{ij}(0) \cap S_{ij} \neq \emptyset, \tag{2.2}$$

$$\mathcal{L}_{ij}(t) \subseteq Inv_{ij}, \quad \forall t \in [0, T_{ij}], \tag{2.3}$$

$$\mathcal{L}_{ij}(T_{ij}) \subseteq G_{ij}, \tag{2.4}$$

$$\dot{V}(x,t) < 0, \quad \forall x \in \mathcal{L}_{ij}(t) \cap \partial X_j \neq \emptyset, \quad \forall d \in D, \quad \forall t \in [0, T_{ij}]. \tag{2.5}$$

where condition (2.2) assures that $\mathcal{L}_{ij}$ has a nonempty intersection with the start set. Note that when $X_i = X_j$, condition (2.5) can be dropped, and condition (2.3), the set inclusion Inv_{ij} is merely $\bar{X}_i$.

Extending Controller Coverage: Inward-Facing Reach Tubes, $\mathcal{L}_i^c$

Although in principle it is possible to employ transition controllers to synthesize atomic controllers for a given FSM, in practice, there are cases where it is impossible to find reactively composable sets which are not spatially disconnected, as required to satisfy (2.3). This can happen whenever the computed reach tubes are small compared with the region; for example, when constructing controllers in long corridors or regions with a large number of obstacles. In

similar rationale to the techniques in [29, 82] that employ a maximization step to widen the basin of attraction to a goal region, we introduce another type of reach tube, *inward-facing* reach tubes, to achieve the needed spatial coverage.

Consider a state $q_i \in Q$. Our goal is to generate atomic controllers that admit finite-time trajectories and satisfy the following two conditions:

1. The region X_i must be invariant; that is, trajectories starting within some subset of X_i must remain in X_i.
2. Trajectories need to reach a goal set $G_i^c \subseteq X_i$; that is, $\xi_T(T_i) \in G_i^c$.

The above statements require that trajectories are both invariant to the region and are sequentially composable with respect to G_i^c. This leads immediately to the following reach tube conditions:

$$\mathcal{L}_i^c(t) \subseteq X_i, \quad \forall t \in [0, T_i], \tag{2.6}$$

$$\mathcal{L}_i^c(T_i) \subseteq G_i^c. \tag{2.7}$$

Notice that, by adding in the inward-facing controllers, we are able to expand the reactively-composable invariant in (2.3) (with a slight abuse of terminology) as $Inv_{ij} = (\bar{X}_i \cup \mathcal{L}_i^c) \cup (\bar{X}_j \cup \mathcal{L}_j^c)$, and likewise also expand the goal set in (2.4) as $G_{ij} = \bar{X}_j \cup \mathcal{L}_j^c$. As will be shown in the next section, this additional set of controllers will help our iterative approach for synthesizing atomic controllers. The larger the sets are, the greater the likelihood of finding controllers that satisfy the specification.

2.3.3 Atomic Controller Synthesis Algorithm

Using the composition strategies and the two types of atomic controllers, we now outline our process for constructing atomic controllers in Algorithm 2.1. The basic procedure is as follows. First, the set of all transitions Δ are extracted from FSM A (`automTransitions` in line 2). Next, atomic controllers and their reach tubes are computed for each edge in Δ in lines 8–18. Reach tubes are computed iteratively until either all possible configurations within $R(q_i)$ for each q_i are enclosed (to within a desired tolerance) or until it is determined that coverage is not possible, i.e. it is not possible to compute $\mathcal{L}_{ij}$ for some $(q_i, q_j) \in \Delta$. The algorithm terminates successfully if reach tubes are found for each edge (line 20). If not, then the reach tube computations are revised by repeating lines 7–25 to ensure they are reactively composable in the sense of Definition 2.2. That is, each reach tube associated with either an incoming or outgoing transition from q_i is checked whether or not it lies within the set of states for which all successor regions of q_i are reachable.

Computing $\mathcal{L}_{ij}$

Figure 2.3 illustrates, through an example, the computation steps in Algorithm 2.1. In the first iteration of lines 8–14, reach tubes are computed for each edge $(q_i, q_j) \in \Delta$. The set $\mathcal{L}_{ij}$ is initialized as the whole configuration space, while the goal set G_{ij} is the region $R(q_j)$ and the invariant Inv_{ij} is the region $R(q_i) \cup R(q_j)$. In Figure 2.3(a), reach tubes are computed for the two transitions (q_1, q_2) (blue region) and (q_1, q_3) (green region), and the intersection of the two is taken (yellow region). Intuitively, this intersection (see Figure 2.3(b)) defines the set of states from which any region of successor states can be reached (by using either

Algorithm 2.1: Construct atomic controllers for all execution paths of A.

procedure CONSTRUCTCONTROLLERS((A, R, f, ϵ, N))

Input: Synthesized FSM A with region mappings $R(\cdot)$, closed-loop robot dynamics $f(\cdot)$, coverage metric ϵ, and number of iterations N for coverage

Output: A set of funnels $\mathcal{L}$ and controllers κ guaranteeing the execution of A

$(\Delta, I_{out}) \leftarrow$ `automTransitions`(A)

for $(q_i, q_j) \in \Delta$ **do**

$\mathcal{L}_{ij} \leftarrow \mathbb{R}^n$

end for

$\mathcal{L}_i^c \leftarrow \emptyset, \qquad \kappa_i^c \leftarrow \emptyset \qquad \forall q_i \in Q$

while `True` **do** ▷ *Repeat until all reach tubes are reactively composable or failure*

for $(q_i, q_j) \in \Delta$ **do**

$S_{ij} \leftarrow \bigcap_{k \in I_{out}^i} \mathcal{L}_{ik} \cap R(q_i)$

$G_{ij} \leftarrow \left(\bigcap_{k \in I_{out}^j} \mathcal{L}_{jk} \cap R(q_j)\right) \cup \mathcal{L}_j^c$

$(\mathcal{L}_{ij}, \kappa_{ij}) \leftarrow$ `getReachTube`$(S_{ij}, G_{ij}, S_{ij} \cup \mathcal{L}_i^c \cup G_{ij}, f, \epsilon, N)$

if $\mathcal{L}_{ij} = \emptyset$ **then**

return $\emptyset$ ▷ *No controller exists*

end if

$S_i^c \leftarrow R(q_i) \setminus \bigcap_{k \in I_{out}^i} \mathcal{L}_{ik}$

$G_i^c \leftarrow \bigcap_{k \in I_{out}^i} \mathcal{L}_{ik} \cap R(q_i)$

$(\mathcal{L}_i^c, \kappa_i^c) \leftarrow$ `getReachTube`$(S_i^c, G_i^c, R(q_i), f, \epsilon, N)$

end for

if $\forall (q_i, q_j) \in \Delta : \left[\left(\mathcal{L}_{ij} \cap R(q_i)\right) \subseteq \left(\left(\bigcap_{k \in I_{out}^i} \mathcal{L}_{ik} \cap R(q_i)\right) \cup \mathcal{L}_i^c\right)\right] \wedge$

$\left[\left(\mathcal{L}_{ij} \cap R(q_j)\right) \subseteq \left(\left(\bigcap_{k \in I_{out}^j} \mathcal{L}_{jk} \cap R(q_j)\right) \cup \mathcal{L}_j^c\right)\right]$ **then**

$\mathcal{L} \leftarrow \left(\bigcup_{i,j} \mathcal{L}_{ij} \bigcup_i \mathcal{L}_i^c\right)$

$\kappa \leftarrow \left(\bigcup_{i,j} \kappa_{ij} \bigcup_i \kappa_i^c\right)$

return $\mathcal{L}, \kappa$

end if

end while

end procedure

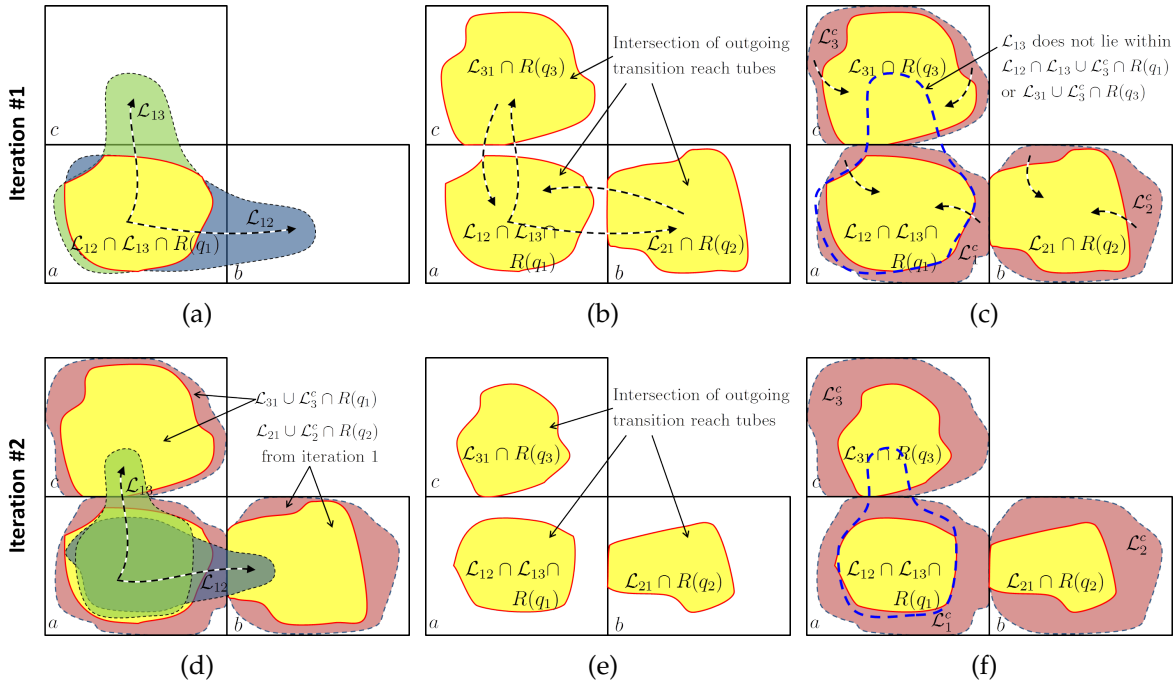


Figure 2.3: Illustration of the reach tube computation steps, assuming symmetric transitions between each adjoining region. In (a), a pair of transition reach tubes $\mathcal{L}_{12}$ and $\mathcal{L}_{13}$ are computed for q_1 , the intersection of which (yellow) defines the new start set for the next iteration (see lines 8–14 in Algorithm 2.1). In (b), the same is done for the remaining states q_2 and q_3. Next, in (c), inward reach tubes $\mathcal{L}_i^c$ (red) are generated for each region, (see lines 15–18 in Algorithm 2.1). This expanded region defines the invariant for the next iteration. In (d)–(f), the process in lines 8–18 is again repeated for the new start sets and invariants, and terminates at (f) since all reach tubes lie inside the regions bounded by the dotted borders (e.g. for q_1 this is $R(q_1) \cap ((\mathcal{L}_{12} \cap \mathcal{L}_{13}) \cup \mathcal{L}_1^c)$).

controller κ_{12} or κ_{13}). The process repeats for the remaining edges in the FSM. The algorithm immediately returns failure if an edge is encountered where a reach tube cannot be constructed (lines 12–14).

Computing $\mathcal{L}_i^c$

In order to expand the size of reactively composable regions, we create inward reach tubes in lines 15–18 to provide controllers that steer the robot to a configuration from which it can take a transition. The collection of $\mathcal{L}_{ij}$ from the current iteration produce the start sets S_i^c and the sequentially composable goal sets G_i^c for each q_i. The set S_i^c in line 15 is the set $R(q_j)$ minus the intersection of all transition reach tubes from that region (the white portions in Figure 2.3(b)), while the set G_i^c in line 16 is found from Definition 2.1 (the yellow portions in Figure 2.3(b)).

In Figure 2.3(c), the red regions enclosed by the dashed lines, $\mathcal{L}_i^c$, denote where controllers were found to drive the system into the yellow region. We seek transition reach tubes that are contained within the union of the red and yellow regions in Figure 2.3(c).

Further Iterations

After a single iteration, if the sequentially-composable transition reach tubes are not reactively composable, the algorithm continues alternately computing $\mathcal{L}_{ij}$ and $\mathcal{L}_i^c$ until they are reactively composable for all $\mathcal{L}_{ij}$. To test if $\mathcal{L}_{ij}$ is reactively composable, we need to determine if $\mathcal{L}_{ij}$ is contained within a subset of states where outgoing transitions from q_i or q_j are possible, i.e. satisfies $\left(\mathcal{L}_{ij} \cap R(q_\alpha)\right) \subseteq \left(\left(\bigcap_{k \in I_{out}^\alpha} \mathcal{L}_{\alpha k} \cap R(q_\alpha)\right) \cup \mathcal{L}_\alpha^c\right)$ for $\alpha \in \{i, j\}$. As such, the termination criterion in line 20 enforces Definition 2.2, by requiring that transition reach tubes must either lie within an inward reach tube or the sets where any successor state is reachable. An additional iteration of the algorithm is shown pictorially in the bottom row

of Figure 2.3.

In any given iteration, the sets S_{ij}, G_{ij}, and Inv_{ij} for the ijth edge are updated by the $\mathcal{L}_{ij}$ and $\mathcal{L}_i^c$ from the previous iteration. Figure 2.3(d) shows the second iteration of lines 8–14, where new transition reach tubes for a are computed ($\mathcal{L}_{12}$ and $\mathcal{L}_{13}$), constrained to stay within the red and yellow regions for q_1, q_2, and q_3. After intersections are taken (yellow regions in Figure 2.3(e)), the reach tubes from the previous iteration are replaced with a new set of inward reach tubes computed in lines 15–18. Figure 2.3(f) illustrates this last step, and is an example of a situation where the algorithm successfully terminates because the reactive composability criterion in line 20 is fulfilled. If the algorithm terminates successfully, a library of reach tubes $\mathcal{L}$ is returned in lines 22–20 along with a library of controllers C.

2.4 Computing Atomic Controllers

We now present an implementation for constructing reach tubes in possibly cluttered workspaces. Probabilistic planning approaches, such as rapidly-exploring random trees [54], have gained widespread use in various path planning applications. In such methods, exploration of the configuration space is probabilistically complete; that is, the probability of solving a motion planning problem for a given initial configuration improves with the number of samples. Recent techniques such as the Invariant Funnels technique in [82] uses sampled trajectories to compute verified controllers in a randomized tree structure. We adopt this framework to construct, in a piecewise manner, the reach tubes in Algorithm 2.1. For each computed sample trajectory, we also compute a region of

invariance (funnel) about the sample trajectory. The workflow for constructing each funnel is as follows [62, 82]: (1) generate a nominal trajectory connecting a given starting configuration and goal configuration, (2) design a local feedback controller to stabilize about the trajectory, and (3) solve a sum-of-squares program for the trajectory/controller pair to find the maximally-permissive funnel for this trajectory.

Throughout this section, let us denote m as the index of a sample trajectory associated with some reach tube $\mathcal{L}$. Let ℓ^m and $\boldsymbol{\kappa}^m$ denote, respectively, a funnel and controller associated with this trajectory. Each reach tube $\mathcal{L}$ is constructed from a collection of funnels such that $\mathcal{L} = \cup_m \ell^m$. Likewise, each atomic controller κ is constructed from a collection of local controllers such that $\kappa = \cup_m \boldsymbol{\kappa}^m$.

2.4.1 Trajectory Generation

In our work, we solve a two-point planning problem to generate each sample trajectory, where we attempt to connect a point in S with a goal point in G. For differentially-flat platforms such as nonholonomic wheeled mobile robots, we apply feedback linearization [70]: a nonlinear transformation on the robot's inputs yielding new pseudo-inputs that are derivatives of the robot's Cartesian coordinates. For static feedback linearization the pseudo-inputs are $u_{pseudo} = [\dot{x}, \dot{y}]^T$. We can then generate an instantaneous steering command by choosing this command to be the Cartesian vector displacement between the current robot configuration and the desired goal configuration, i.e. $u_{pseudo} = [x - x_{goal}, y - y_{goal}]^T$. Using this strategy, we make use of standard ODE solvers to solve the planning problem.

We discard those trajectories that do not satisfy the constraints for the current reach tube (as detailed in Section 2.3.2); those that are accepted are represented by the pair (ξ_T^m, μ_T^m). For systems which are not feedback linearizable (e.g. 3-D Cartesian robot arms), trajectories can still be generated using nonlinear trajectory optimization methods [9], or any number of motion planning tools.

2.4.2 Trajectory-Stabilizing Controllers

We apply local controllers to correct for deviations from the nominal sample trajectory due to disturbances or initialization errors. In this work, we use a linear quadratic regulator (LQR) approach [44] applied to a linearized version of the system (2.1) based on the mth trajectory (ξ_T^m, μ_T^m). LQR controllers are generated using the metric quantities $\bar{x}(t) = x(t) - \xi_T^m(t)$ and $\bar{u}(t) = u(t) - \mu_T^m(t)$ using a cost function of the form $\int_0^T (\bar{x}^T Q_1 \bar{x} + \bar{u}^T Q_2 \bar{u}) dt + \bar{x}^T S_T \bar{x}$. The matrices Q_1, Q_2, and S_T are design parameters that can be adjusted to tailor the shape of the funnels. In our case, we are interested in funnels with wide mouths (initial sets) and small tails (goal sets), so we typically choose S_T one order of magnitude larger than Q_2, while the values in Q_1 are chosen to remain within the same relative order as Q_2. The control gain $K^m(t)$ is computed based on the matrix solution $S^m(t)$ to a Riccati equation.

2.4.3 Invariant Funnels

Given the nominal trajectory (ξ_T^m, μ_T^m) and controller $K^m(t)$, the task is to find a maximally-permissive funnel that satisfies the conditions in Section 2.3.2.

The matrix solution to the Riccati equation from the controller generation step immediately parameterizes quadratic Lyapunov functions $V^m(x,t) = \bar{x}^T \mathcal{S}^m(t)\bar{x}$. Working with quadratic functions averts the problem of searching exhaustively over all classes of Lyapunov function candidates. Given that quadratic representations in general carry only local guarantees, the task is equivalent to finding a maximal $\rho^m(t)$ such that the $\rho^m(t)$-sub-level sets of these Lyapunov functions $\ell^m(\rho^m(t),t)$ (represented as ellipsoids) satisfy the finite-time invariance conditions explained in [82], while also adhering to the conditions of Section 2.3.2.

To explain our approach in the context of [62, 82], we present the most general problem statement for solving for transition reach tubes using the system (2.1), then remark on which parts of the problem are changed when adapting the problem for inward-facing reach tubes. Computing a transition reach tube involves the following objective:

$$\max \rho^m(t), \qquad t \in [0, T^m] \tag{2.8}$$

$$\text{s.t.} \qquad \rho^m(t) \geq 0, \qquad \forall t \in [0, T^m], \tag{2.9}$$

$$\dot{V}^m(x,t) \leq \dot{\rho}^m(t), \qquad \forall t \in [0, T^m], \forall x \in \{x \mid V^m(x,t) = \rho^m(t)\}, \forall d \in D \tag{2.10}$$

$$\dot{V}^m(x,t) \leq 0, \qquad \forall t \in [0, T^m], \forall x \in \{x \mid V^m(x,t) = \rho^m(t)\} \cap X_j, \tag{2.11}$$

$$\ell^m(\rho^m(t),t) = \{x \mid V^m(x,t) \leq \rho^m(t)\} \subseteq Inv, \qquad \forall t \in [0, T^m], \tag{2.12}$$

$$\ell^m(\rho^m(T^m),T^m) = \{x \mid V^m(x,T^m) \leq \rho^m(T^m)\} \subseteq G \tag{2.13}$$

where X_j is the polyhedral set corresponding to the successor state q_j. Constraints (2.9) and (2.10) enforce trajectory invariance to the funnel and positive-semidefiniteness of the level set. The constraint (2.11) is a re-statement of (2.5) in the funnel setting. Likewise, (2.12) and (2.13) are, respectively, re-statements of the invariance (2.3) and goal (2.4) conditions. Condition (2.2) is not included

here because satisfaction of the start set is implicitly satisfied by sampling the initial state of the trajectory from within S. It is worth pointing out that the difference between our formulation and the one in [62] is precisely the addition of conditions (2.11)–(2.13).

Oftentimes, we find there are unknown disturbances (wind gusts pushing the robot in one direction or another), causing unexpected motions and collisions with obstacles. When we have such disturbances affecting the dynamics, we require that the invariance condition in (2.9) to be true for *all* $d \in D$.

When computing funnels for inward reach tubes, the inequality (2.11) is removed and the inclusion (2.12) is replaced with the following:

$$\ell^m(\rho^m(t), t) = \{x \mid V^m(x,t) \leq \rho^m(t)\} \subseteq X_i$$

to reflect the condition in (2.6).

When solving the optimization problem in (2.8) – (2.13) using numerical methods, we replace the trajectory (ξ_T, κ_T) with its discrete sequence $(\mathbf{t}^m, \boldsymbol{\kappa}^m, \mathbf{x}^m)$. We then express the closed-loop system $\hat{f}(x,d)$ under the action of $\boldsymbol{\kappa}^m$ as being *polynomial* in its arguments x and d. We then transform the problem (2.8)–(2.13) into a sum-of-squares program replacing the intervals $[0, T^m]$ with $\mathbf{t}^m$ and by making the following substitutions:

Eq. (2.10) **and** (2.11)**:**

$$\begin{cases} -\Big(\frac{\partial}{\partial x}V^m(x,t)\hat{f}(t,x,0) + \frac{\partial}{\partial t}V^m(x,t) + \lambda_1(x,t)\,(\rho^m - V^m(x,t)) \\ \qquad\qquad +\lambda_2(x,t)P_j(x) + \lambda_3(d,t)P_d(d)\Big) \quad \text{is s.o.s.}, \qquad t \in \mathbf{t}^m, \\ \lambda_1(x,t), \;\; \lambda_2(x,t), \;\; \lambda_3(x,t) \;\; \text{are s.o.s.}, \quad t \in \mathbf{t}^m, \end{cases} \tag{2.14}$$

$$\textbf{Eq. (2.12):} \quad \begin{cases} (V^m(x,t) - \rho^m) - \lambda_4(x,t)P_{inv}(x) \quad \text{is s.o.s.}, \quad t \in \mathbf{t}^m \setminus T^m, \\ \lambda_4(x,t) \quad \text{is s.o.s.}, \quad t \in \mathbf{t}^m \setminus T^m, \end{cases} \tag{2.15}$$

$$\textbf{Eq. (2.13):} \quad \begin{cases} (V^m(x,T^m) - \rho^m) - \lambda_5(x)P_G(x) \quad \text{is s.o.s.}, \\ \lambda_5(x) \quad \text{is s.o.s.} \end{cases} \tag{2.16}$$

Where $\lambda_i(x,t)$, $i = 1,\ldots,5$ are positive-definite polynomial multipliers and $P_j(x)$, $P_{inv}(x)$,$P_G(x)$, and $P_d(d)$ are all polynomials used in parameterizing the sets X_j, Inv, G, and D as zero-sub-level sets. In particular,

$$X_j = \{x \mid P_j(x) \leq 0\}, \quad Inv = \{x \mid P_{inv}(x) \leq 0\},$$
$$G = \{x \mid P_G(x) \leq 0\}, \quad D = \{d \mid P_d(d) \leq 0\}.$$

The sum-of-squares program is solved via the MATLAB toolboxes SPOT [66] and Ellipsoids [53].

2.4.4 Algorithm

The steps for generating reach tubes are encapsulated in an algorithm `getReachTube`, outlined in Algorithm 2.2. The process is iterated up to N times. `getInit` on line 4 randomly picks an initial point in the S. The function `getFinal` on line 5 picks a final point inside G, seeking the centroid of the region if the goal set is a polygon and randomly if it is defined by reach tubes. With the boundary conditions defined, the algorithm next invokes `simTrajectory` to generate a feasible trajectory and a controller based on the LQR design. Given a $\mathbf{t}^m$, the function returns a controller library $\boldsymbol{\kappa}^m$ as the sequence $(\boldsymbol{\mu}^m, \mathbf{K}^m)$, where $\boldsymbol{\mu}^m = \{\mu^m(t)\}_{t\in\mathbf{t}^m}$ and $\mathbf{K}^m = \{K^m(t)\}_{t\in\mathbf{t}^m}$. A trajectory is deemed infeasible if it ever leaves the invariant for the current transition. If this happens, new final points and trajectories are generated until it is unobstructed. If a feasible trajectory

is found, `computeFunnels` computes the funnels according to the procedure in Section 2.4.3. Lines 10–13 check if a feasible funnel is found. If so, it is appended to the existing library of funnels.

Algorithm 2.2: Computing reach tubes for a system f respecting state invariants and start and goal sets.

1: **procedure** GETREACHTUBE($(S, G, Inv, f, \epsilon, N)$)

Input: A start set S, a goal set G, an invariant set Inv, along with f, ϵ, N

Output: A set of reach tubes $\mathcal{L}$ and controllers κ

2: $m \leftarrow 0$, $\mathcal{L} \leftarrow \emptyset$, $\kappa \leftarrow \emptyset$

3: **while** $Vol(\mathcal{L} \cap S) < (1 - \epsilon)Vol(S) \wedge m < N$ **do**

4: $m \leftarrow m + 1$

5: $x^i \leftarrow$ `getInit`(S)

6: $x^f \leftarrow$ `getFinal`(G)

7: $(\mathbf{t}^m, \mathbf{x}^m, \boldsymbol{\kappa}^m) \leftarrow$ `simTrajectory`(f, x^i, x^f, Inv)

8: **if** $\mathbf{x}^m \neq \emptyset$ **then**

9: $\ell^m \leftarrow$ `computeFunnel`$(\mathbf{t}^m, \mathbf{x}^m, \mathbf{c}^m, G, Inv)$

10: **if** $\ell^m \neq \emptyset$ **then**

11: $\mathcal{L} \leftarrow \mathcal{L} \cup \ell^m$

12: $\kappa \leftarrow \kappa \cup (\mathbf{t}^m, \mathbf{x}^m, \boldsymbol{\kappa}^m)$

13: **end if**

14: **end if**

15: **end while**

16: **return** $\mathcal{L}, \kappa$

17: **end procedure**

Note that perfect coverage of a region is often not possible when the boundaries of a region are included as constraints. We allow for incomplete cover-

age by introducing a coverage metric $\epsilon \in [0, 1]$ and declare the set covered if $Vol(\mathcal{L} \cap S) \geq (1 - \epsilon)Vol(S)$ or if $m = N$ where Vol is the volume of a particular set defined in $\mathbb{R}^n$, m is the current funnel iterate, and N is an integer. Since the set S is represented as the intersections of unions of ellipsoids, the volume is computed in an approximate manner with the aid of the Ellipsoids toolbox [53]. The former condition asserts that coverage terminates if the volume of the reach tube $\mathcal{L}$ within the start set is a significant enough fraction of start set. If the coverage is not achieved before N iterations, then there may be transitions from region $\gamma_{\mathcal{R}}(q_i)$ that are not reachable from some parts of the state space $R(q_i)$ for that region. Hence, the method is sound, but not complete.

Another implementation issue arises from the curvature of the ellipsoidal level sets. At the boundaries of polyhedral sets, coverage degenerates where the ellipsoid level sets meet the polyhedral region boundaries, sometimes causing difficulty when generating transition reach tubes. We work around this issue by relaxing the interface between neighboring regions by some fixed tolerance value ϵ_{reg}. This parameter allows regions to share territory by an amount defined by this fixed distance. To more strictly enforce one of the two region boundaries, one can adjust the shared boundary in the direction of one region or another. Although not explored in this work, it is also possible to remove this shared boundary by altering our approach slightly to allow ellipsoidal level sets to be expanded beyond region boundaries as long as the boundary itself is certified as an invariant.

2.4.5 Complexity

Here we state the time complexity of our overall approach. The implementation in Algorithm 2.1 runs in time $O(|Q|^3)$. Put in terms of a set number of transitions $|\Delta|$, we can express this more precisely as $O(|\Delta||Q|)$. Algorithm 2.2 applies two semidefinite programs; one for solving the sums-of-squares program and another for estimating the set volume. Both run in polynomial-time: the sums-of-squares solver is polynomial in both the state dimension n and the size of the FSM $|Q|$, while solver used for volume estimation is polynomial in n. In Section 2.6 we give more empirical insight into the complexity of the approach as a whole by providing actual runtimes when performing the computations for several example tasks.

2.5 Controller Execution

At runtime, low-level controllers are selected according to the current state of the robot and the current values of the sensor propositions. The planner executes the controller associated with the funnel associated with the current robot state (e.g. $\boldsymbol{\kappa}_{ij}^m$ if within $\ell_{ij}^m(t)$). In order to adhere to the transitions of the controller FSM as the system evolves, a new funnel is selected if one of three events occur: (i) the end of a funnel is reached, (ii) a region transition is made and either an inward funnel or a transition funnel for the next transition is reached, or (iii) an environment proposition changes. Priority is always given to transition controllers κ_{ij} over inward ones κ_i^c. That is, if the robot is currently executing a controller in κ_i^c and the trajectory reaches a funnel in $\mathcal{L}_{ij}$, with r_j as the goal for that transition, the motion controller is switched from κ_i^c to κ_{ij}. In Example 2.1,

consider the case when the robot is in state q_4 with $S\_blocked$ = `False`. At the current time step, the robot is executing the controller κ_{34} and has just reached O. The next goal (S) is implemented by switching controllers to κ_{41} if within the associated funnel. Otherwise, the planner will choose κ_4^c. If the trajectory is in more than one funnel, the execution paradigm operates deterministically (the first funnel encountered in the library is always selected).

We now show that the algorithm, when executed using the runtime implementation described above, produces trajectories that remain consistent with respect to the behaviors in the FSM. Define $\omega : \mathbb{R}_+ \to \mathcal{X}$ as a possible environment proposition trajectory initialized at $\omega(0)$. Let $q(0)$ be the initial state, and $\xi^{q(0)}$ be a disturbance-free continuous trajectory in $X_{q(0)}$, with the robot's configuration initially $\xi^{q(0)}(0)$. Let $q(k)$, $k > 0$ be defined as follows. Initializing $k = 0$, $\tau_0 = 0$, we increment k either when a time $t = \tau_k > 0$ is reached for which $\xi^{q(k)}(\tau_k) \in \partial X_{q(k)}$ (the trajectory reaches a boundary) or when $\delta(q(k), \omega(\tau_k)) = q(k)$ (a self-transition is already satisfied at time $t = \tau_k$ under the environment input $\omega(\tau_k)$). Each time k is incremented, let $\xi^{q(k-1)}(\tau_{k-1}) = \xi^{q(k)}(\tau_{k-1})$ to ensure continuity of the trajectories and build a trajectory segment $\xi^{q(k)}(t)$, $t \in [\tau_{k-1}, \tau_k)$. To avoid livelock, we assume that the environment does not change too fast by restricting $\omega(t)$, $t \in [\tau_k, \tau_{k+1})$ to only those input traces for which livelocking does not occur for all $k \in \mathbb{N}$. Given any such ω, we call the sequence $\sigma = \omega(0)\mathcal{R}(q(0))\omega(\tau_1)\mathcal{R}(q(1))\ldots$ a *reactive execution trace* associated with the continuous trajectory.

In the following proposition, we show that the continuous trajectories obtained by executing the atomic controllers synthesized from Algorithm 2.1 yield reactive execution traces that enforce the behaviors of the high-level controller.

Proposition 2.1. *Consider a robot initialized within* $\bigcap_{k \in I^i_{out}} \mathcal{L}_{ik} \cup \mathcal{L}_i^c$, $q_i \in Q_0 \wedge (q_i, \cdot) \in \Delta$.

For all ω, executing Algorithm 2.1 produces a reactive execution trace σ of the high-level controller A.

Proof. To show that σ produced by the execution is in fact an execution trace of A, we remark that reach tubes are constructed from edges in Δ. For any $(q_i, q_j) \in \Delta$, the robot will either be in $\mathcal{L}_{ik}$ for some $k \in I^i_{out}$ where conditions (2.3) and (2.4) hold true, or will be in $\mathcal{L}^c_i$, where (2.6) and (2.7) hold true. Once in X_j, the robot will not re-enter X_i as a consequence of (2.5). Therefore, the robot will not exhibit additional behaviors not in A.

To show that the converse is true; that is, there exists a σ for every possible execution path of A, it is only necessary to show that, for any state in $\bigcap_{k \in I^i_{out}} \mathcal{L}_{ik} \cup \mathcal{L}^c_i$, we can take any valid transition from q_i. By construction, the set $\bigcap_{k \in I^i_{out}} \mathcal{L}_{ik} \cup \mathcal{L}^c_i$ is reactively composable, thus proving completeness of the executions of A. □

Note that in general possible successor regions can be far away from each other and so, we cannot guarantee the behaviors of A for any arbitrary trajectory of environment propositions. For example, consider again state q_4 in Example 2.1. The environment is allowed to toggle *S_blocked* between `True` and `False` indefinitely when the robot is within O. In the continuous setting, this would cause the robot to get trapped forever in q_4, while in the FSM, there is no such livelock, since this toggling does not appear in the discrete execution. This is due to the physical setting in which robots operate and is a limitation of the abstraction, rather than the low-level controller synthesis approach.

2.6 Simulations

In this section, we demonstrate the application of the method developed in this chapter to three examples. For these examples, we make use of a unicycle robot model consisting of three states, governed by the kinematic relationship:

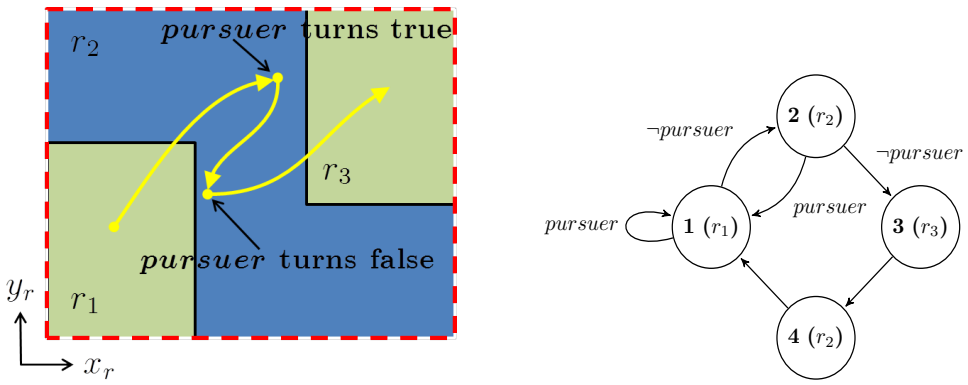
$$\begin{bmatrix} \dot{x}_r \\ \dot{y}_r \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} v\cos\theta \\ v\sin\theta \\ \omega \end{bmatrix},$$

where x_r and y_r are the Cartesian coordinates of the robot, θ is the orientation angle, and v and ω are, respectively, the forward and angular velocity inputs to the system. In this work, we augment the model by limiting ω such that

$$\omega = \begin{cases} \omega_{max} & \text{if } u \geq \omega_{max} \\ \omega_{min} & \text{if } u \leq \omega_{min} \\ u & \text{otherwise} \end{cases}$$

and $v = v_{nom}$. The input u is governed by a feedback controller that steers the robot from some initial configuration to the desired configuration. We adopt the parameter settings $\omega_{min} = -3$, $\omega_{max} = 3$, and $v_{nom} = 2$.

For each example, we perform computations using MATLAB on a standard 64-bit Windows desktop machine, with an Intel Core i7 processor clocked at 3.40 GHz and 8 GB of RAM. To simplify our implementation, when computing funnels we make sure to choose from a single ρ^m regardless of the time index, so that $\dot{\rho}^m = 0$.



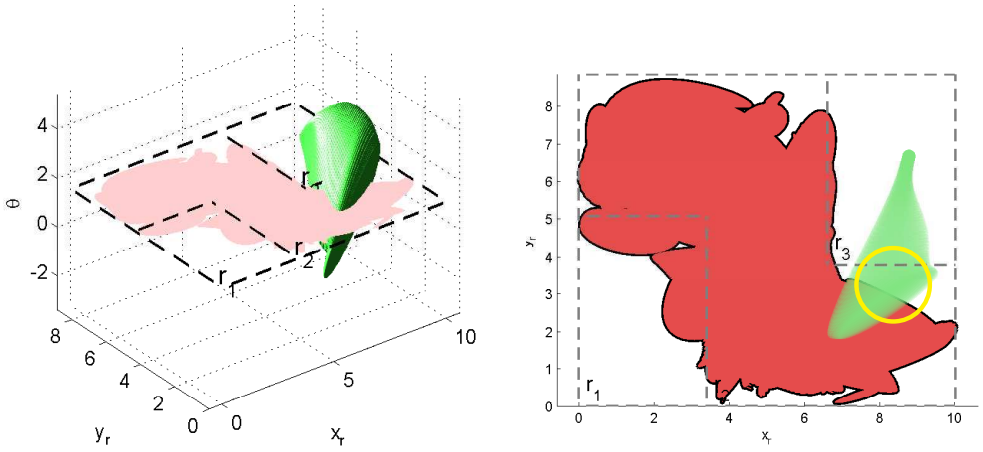
(a) (b)

Figure 2.4: (a) Workspace for the "patrolling two regions" example. (b) FSM for the example. The transitions are labeled with the truth value of the *pursuer* sensor; unlabeled transitions imply that *pursuer* can take on any value.

2.6.1 Patrolling Two Regions

Consider a unicycle robot moving in the $10m \times 9m$ workspace shown in Figure 2.4(a). The robot is initially in r_1 and must continually patrol r_3 and r_1. If the robot senses a pursuer, then it must return to r_1 (safe zone). The specification is implemented in the FSM shown in Figure 2.4(b).

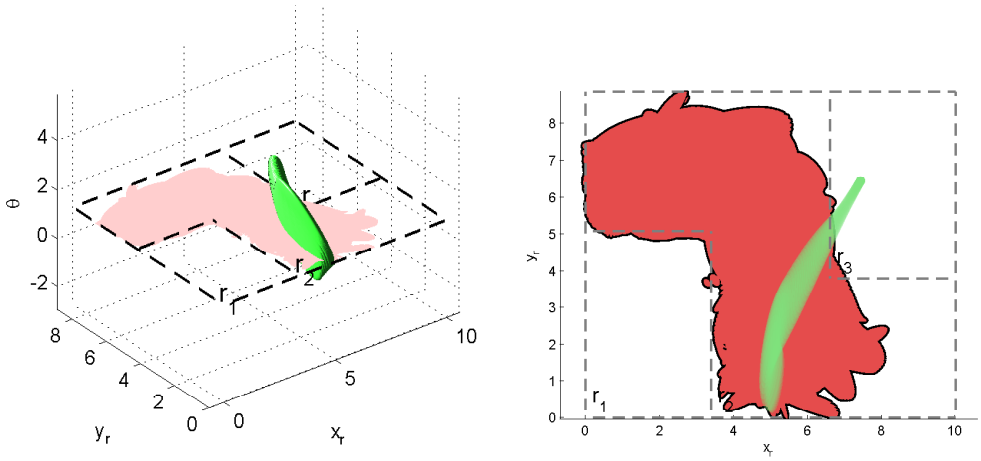
A library of reactively-composable controllers is generated. For this example, we set the coverage metric $\epsilon = 0.2$, the number of iterations $N = 100$, and the interface relaxation $\epsilon_{reg} = 0.2m$. Reach tubes $\mathcal{L}_{23}$ are generated in the first and second iterations for (r_2, r_3) and sample funnels are shown in Figs. 2.5 and 2.6. Also shown are the θ-slices of the union of the inward reach tube and transition reach tube $\mathcal{L}_2^c \cup \mathcal{L}_{21}$ (shown in red). In the first iteration, it is apparent that the funnel spans a gap in the set of inward funnels and hence the controllers are not reactively composable. If the controller drives the robot to this portion of the configuration space, the robot cannot change direction if it senses a pursuer.



(a) (b)

Figure 2.5: (a) shows a transition funnel for the transition from r_2 to r_3 and a slice of the set where r_1 is reachable from r_2 at $\theta = 1.35$ (defined by $\mathcal{L}_2^c \cup \mathcal{L}_{21}$) after the *first* iteration of Algorithm 2.1. (b) shows a 2-D view of the slice. In this iteration, the funnel is not reactively composable because the portion in r_2 is not confined to the red set. If the robot is outside of the red set when enroute to r_3, there are no controllers which can deliver it to r_1 if it sees a pursuer.

We thus continue with another iteration of Algorithm 2.1. After the second iteration, the revised funnel is reactively composable for all transitions and no further iterations are necessary. The volumetric region coverage for each of the four states are shown in Table 2.1. Here, volume fraction is defined as the ratio of the actual volume of the region polytope $R(q_i)$ in (x_r, y_r, θ) and the subset of that polytope which contains $\bigcap_{k \in I_{out}^i} \mathcal{L}_{ik} \cup \mathcal{L}_i^c$.



(a) (b)

Figure 2.6: (a) shows a transition funnel for the transition from r_2 to r_3 and a slice of the set where r_1 is reachable from r_2 at $\theta = 1.05$ (defined by $\mathcal{L}_2^c \cup \mathcal{L}_{21}$) after the *second* iteration of Algorithm 2.1. (b) shows a 2-D view of the slice. The funnel is now reactively composable because the portion of it which lies in r_2 is now completely enclosed by the red set. The robot is therefore able to move to r_1 if it senses a pursuer anywhere along its path to r_3.

Table 2.1: Fraction of (x_r,y_r,θ) coverage for each X_i associated with the FSM.

q_1	q_2	q_3	q_4
(r_1)	(r_2)	(r_3)	(r_2)
0.5846	0.5976	0.5295	0.6171

Figure 2.7(a) shows a sample trajectory of a robot starting in r_1, in which a controller in κ_{12} is applied, followed by a controller in κ_2^c and one in κ_{23}. Part way through its motion to r_3, the *pursuer* sensor turns `True`, invoking the sequence κ_2^c, κ_{21} to take the robot to r_1. *pursuer* once again becomes `False` prompting activation of a controller in κ_2^c followed by one in κ_{23}. As can be seen, the robot remains within the funnels and is able to satisfy the specification regardless of

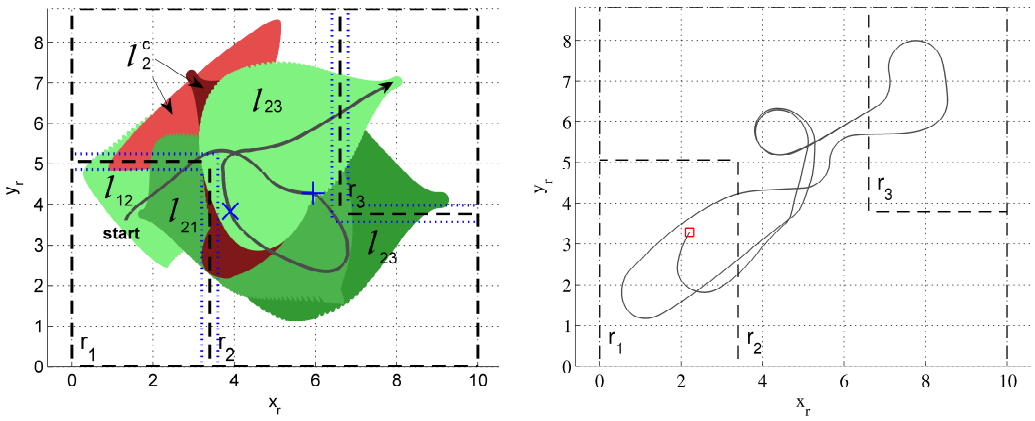


Figure 2.7: Closed-loop trajectory generated from an initial state in r_1. (a) shows a set of control laws are applied to implement the transitions (r_1, r_2) and (r_2, r_3), driving the robot from state 1 to state 2, then from state 2 to state 3 in Figure 2.4(b). 2-D projections of the active funnels are also shown, the red corresponds to inward and green corresponds to transition. *pursuer* turns `True` when at the location marked by the "+" sign. At this instant, another controller is invoked to make the transition (r_2, r_1). *pursuer* turns `False` at the "×" location, and new control laws are used to resume the transition (r_2, r_3). (b) shows an long-term path when the robot starts at the red "□" in r_1 and *pursuer* remains `False` throughout.

the environment as it moves between regions. Figure 2.7(b) shows a long-term trajectory starting at a given initial condition in $r_1 \cap ((\mathcal{L}_{12} \cap \mathcal{L}_{11}) \cup \mathcal{L}_1^c)$. The figure indicates that the controllers generated by our algorithm can produce an infinite trajectory which is correct with respect to the infinite execution traces of A.

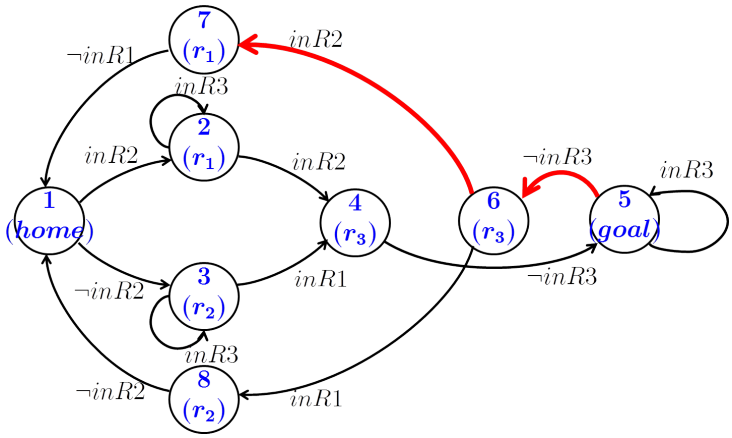


Figure 2.8: FSM for the pursuit-evasion example. For simplicity, we label each edge by the value the sensor proposition must take, and exclude labels for which the remaining input labels can be inferred based on the assumptions on the enemy's behavior. For example, the transition (q_6, q_7) is labeled *inR*2; by mutual exclusion of the regions the enemy can occupy, when *inR*2 is `True`, *inR*1 and *inR*3 are `False`.

2.6.2 Pursuit-Evasion Game

Consider a game being played between a robot and an adversary, where the robot must repeatedly visit the home and goal regions pictured in Figure 2.9, while evading an adversary which visits each of the three remaining regions infinitely often. Evasion is encoded by the requirement that the robot should never enter the same region as the enemy. By assuming the enemy patrols its three regions, the robot cannot get trapped forever in the goal region. As a fairness condition, in the specification we assume also that the enemy cannot enter the region the robot is currently in. The high-level controller (Figure 2.8) consists of eight states and 13 transitions. The sensor values *inR*1, *inR*2 and *inR*3 correspond to the sensed location of the adversary.

Reach tubes are constructed for each of the 13 transitions. A subset of these

(the highlighted edges in Figure 2.8) are shown in Figure 2.9, showing the possible trajectories that the robot may follow when transitioning between *goal* and r_3 (denoted green), and when transitioning between r_3 and r_1. Note that the *goal*–r_3 funnels all deliver the robot to the left of *goal*. The reason for this is that there are two possible transitions out of r_3 (r_1 and r_2) depending on the location of the enemy. If the robot invokes these reach tubes, it always exits the left-hand facet of the goal region, where the robot is easily able to toggle between the goals r_1 and r_2 as the enemy toggles between those regions. The gray trajectory in Figure 2.9 illustrates this for the case when *inR*1 remains `True`. Without reactive composition, a motion planner may cause the robot to exit *goal* via the top or bottom facets. If the robot follows the hypothetical magenta trajectory in Figure 2.9, if *inR*1 stays `True` when in r_3, the robot will have no other option but to enter r_1 (violating the specification) because there are neither any transition funnels $\mathcal{L}_{68}$ nor any inward funnels $\mathcal{L}_6^c$ in the area above the goal region.

2.6.3 Delivery in a Cluttered Environment

We next return to the example in Example 2.1. The task, once again, is to repeatedly deliver supplies between region S and R, connected via O, and return to C if the store S is blocked. We apply two models to fulfill this task: a model of the under-actuated unicycle and a model of a non-holonomic car. The non-holonomic car is represented by the four-dimensional system:

$$\begin{bmatrix} \dot{x}_r \\ \dot{y}_r \\ \dot{\theta} \\ \dot{\phi} \end{bmatrix} = \begin{bmatrix} v\cos\theta \\ v\sin\theta \\ \frac{v}{\ell_r}\tan\phi \\ \omega \end{bmatrix},$$

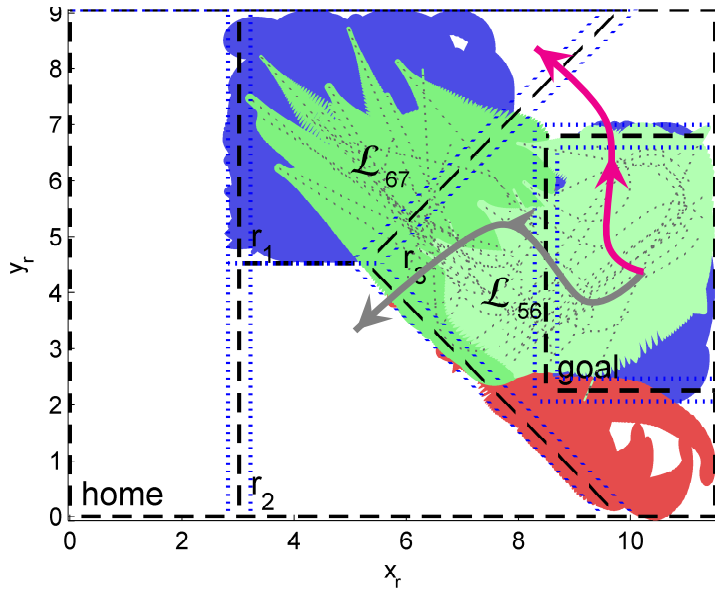


Figure 2.9: Transition funnels for $\mathcal{L}_{56}$ and $\mathcal{L}_{67}$ (green) in the pursuit-evasion example. $\mathcal{L}_5^c$ and $\mathcal{L}_7^c$ are shaded blue and $\mathcal{L}_6^c$ is shaded red. Two hypothetical trajectories are shown; the one exiting the top of goal is not reactively composable and hence always enters r_4 regardless of the environment when in r_3, while the one exiting the left face of goal is reactively composable allowing the robot to choose between entering r_1 or r_2.

where x_r, y_r, θ, v and ω are the same as for the unicycle, ϕ is the steering angle and ℓ_r is the axle spacing, which we take to be $\ell_r = 0.5$ m. In this car model, we have control of both forward velocity v and steering rate ω, and impose no restrictions on their values.

We apply Algorithm 2.1 to this problem for a workspace similar to the one shown in Figure 2.1(a). Because the central region (O) consists of a large area interspersed with several obstacles, we forego attempting to cover the entirety of the region and instead seed inward controllers at configurations where transition controllers already reside. We do this to aid composition; for a given number of funnels, concentrating funnels together reduces gaps in the reactively-

and sequentially-composable sets. Rather than keeping the goal region the same for all inward funnels, we attempt to increase coverage depth by permitting inward funnels to be sequentially composed with one another. We do this by redefining the goal region to include the most recent inward funnel each time one is computed. We limit the number of transition controllers to 20 and the number of inward controllers to 100. With these parameters, each major iteration in the while loop (lines 7–25) the MATLAB implementation took approximately 780 minutes to complete for the unicycle robot and 1470 minutes for the car. For both models, we terminate after two major loop iterations. We note that the method we introduce is intended as an offline verification and controller synthesis approach, and therefore the code and computer hardware implementations were not optimized in this work.

As constructed, the controllers are able to handle infinite executions of the high-level controller in Figure 2.1(b). A partial sample trajectory for the unicycle robot starting in region S is shown in Figure 2.10. The controller coverage (projection of the inward and transition reach tubes onto x_r, y_r) is indicated by the shaded regions in the map. In the first iteration of the synthesis algorithm, transition funnels initially cover a large portion of region O. After the first iteration, sampling of the funnel trajectories becomes more concentrated only in areas where reactive composition can be met. In our case, this is in the central part of the region. A side benefit to the approach is that the resulting controllers tend to yield efficient (non-circuitous) trajectories in the free configuration space. Figure 2.11 shows the same scenario as Figure 2.10, except using the car robot. Similar to the unicycle, the car robot is able to fulfill the transitions of the high-level controller. One of the differences is that the coverage area for the car is slightly smaller than the unicycle, which is primarily due to the

dynamics being restricted by nonholonomic constraints.

For both robot models, the trajectory remains within the verified reach tube for most of the trajectory, but in a few cases the trajectory momentarily exits the reach tube. This is attributable to the fact that we construct reach tubes based on an a polynomial approximation of the true dynamics. We simulate the polynomial approximation (dashed trajectories) in Figures 2.10 and 2.11. For both test cases, we have verified that the trajectories computed with the polynomial approximation do in fact remain within the funnels; this observation is reflected in the figures. If a bound on the divergence between the trajectories of the true system and its approximation can be found, we can use this bound to guarantee correctness in the actual system by imposing a certain amount of inflation on all obstacles and regions. We intend to address such approximation issues as future work.

2.7 Conclusion

In this chapter, a method is presented for synthesizing controllers in the continuous domain based on a discrete controller derived from a high-level reactive specification. The central contribution is an algorithm that generates controllers guaranteeing every possible execution of a finite-state machine for robots with nonlinear dynamics.

Since a large number of computations are required to compute trajectories and funnels satisfying ellipsoidal constraints, the trade-off between completeness and complexity will need to be explored further. In contrast to the approach taken in this work, one could devise a depth-first strategy which seeks to gen-

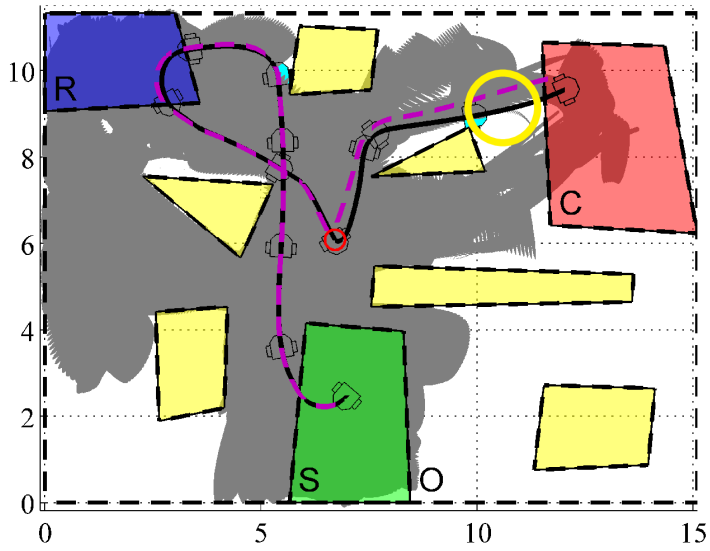


Figure 2.10: Partial sample trajectories of the unicycle robot (solid), and its polynomial approximation (dashed) for the delivery scenario in Example 2.1 . The robot pose is shown at uniform time intervals, and the projection of the funnels appears as shaded fill areas. The red ∘ indicates the rising edge of the *S_blocked* sensor. Note the yellow circled portion of the true system trajectory, indicating a place where it momentarily leaves the funnel.

erate atomic controllers in concentrated parts of the configuration space. While there is much to be gained in terms of computational efficiency (there would be fewer funnels in the database), this would be at the expense of completeness, since the vast majority of possible configurations would not be tied into the funnel libraries.

If a set of atomic controllers cannot be synthesized, a natural question might be to ask: which parts of the specification, when modified, would allow the synthesis of controllers? If the algorithm fails to find a set of low-level controllers satisfying the specification, one may apply techniques such as the one in [6] to revise specifications in such a way that low-level controllers exist. Creating a

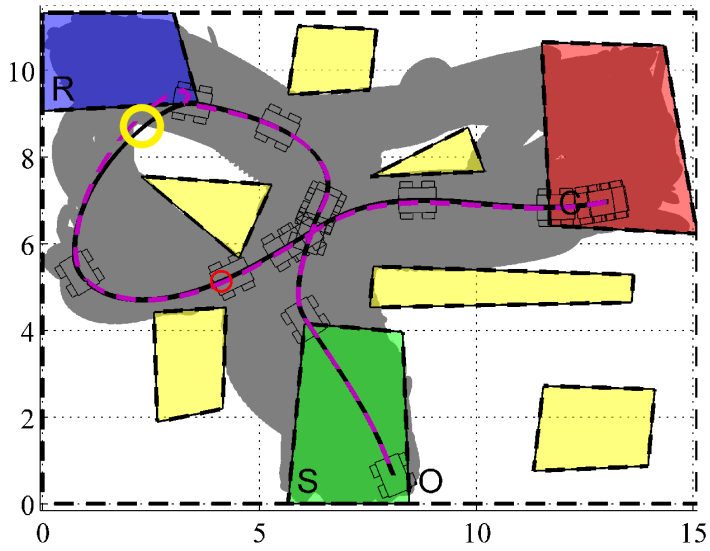


Figure 2.11: Partial sample trajectories of the non-holonomic car (solid), and its polynomial approximation (dashed) for the delivery scenario in Example 2.1 . The robot pose is shown at uniform time intervals, and the projection of the funnels appears as shaded fill areas. The red ∘ indicates the rising edge of the *S_blocked* sensor. Note the yellow circled portion of the true system trajectory, indicating a place where it momentarily leaves the funnel.

general, intuitive way of providing user feedback for specification revisions is a topic we intend to explore as future work.

CHAPTER 3

AUTOMATED GENERATION OF DYNAMICS-BASED RUNTIME CERTIFICATES FOR HIGH-LEVEL CONTROL

3.1 Introduction

As industrial processes, homes, and personal vehicles become more automated, formal approaches to mission planning are particularly appealing as a means to creating reliable controllers. Such controllers find utility when complicated tasks must be carried out in collaborative, human environments in which human safety is a primary concern. Automated synthesis of controllers from high-level specifications also frees users from the burdens of directly programming controllers that satisfy complex tasks. Previous works have focused on tools for automatically synthesizing discrete controllers for carrying out high-level mission plans expressed as formal mission specifications, e.g. [47,52,79,93].

Application of such techniques to physical systems requires the addition of *discrete abstractions* that represent a physical system that must execute the mission (for instance, the dynamics of a mobile robot or a robotic manipulator). Several works have focused attention on computational approaches for obtaining such abstractions that represent a system's dynamics in structured environments; see [39,76,95]. The body of work in computational tools for generating abstractions have enabled others to develop methods for synthesis of mission plans using systems ranging from simple single or double integrators ([36,52]) and piecewise linear models ([84,94]) to nonlinear ([10,39,89,93,95]), switched ([58]) and hybrid systems ([64]). Synthesis for switched systems was considered in [57,58], where the authors propose methods for computing fine-grained

abstractions and switching protocol synthesis for reactive tasks. Tools for automatically synthesizing controllers based on high-level specifications written over task-oriented abstractions of nonlinear systems have been introduced recently in [20].

When there does not exist a controller that is guaranteed to satisfy a specification under the worst-case behaviors of the environment, i.e. it is *unrealizable*, the task of debugging the specification can be difficult and may be worsened with the existence of a discrete abstraction. For instance, if a manipulator tasked with fetching parts on a conveyor belt cannot reach for fast-moving parts, the designer must revise the specification with additional statements that consider both the physics of the manipulator and the underlying assumptions on the uncontrolled environment (in this case, the motion of the parts). To assist the designer, automated frameworks have been introduced recently for debugging unrealizable specifications [49, 73]. Further work has shown progress toward automated repair of unrealizable specifications through the synthesis of revisions to such specifications [5, 34, 55].

In this chapter, we address the problem of realizability of specifications caused by the dynamics of a system by introducing a framework that *automatically* suggests additions to such specifications and provides them to the user in a clear, understandable manner. We focus on systems involving physical motion, encompassing tasks carried out by mobile robots, land or air vehicles, or industrial process machinery, to name a few. We adopt an iterative procedure that uses the discrete abstraction of the physical process to assist in interactively computing revisions to the specification. Using a graphical visualization tool, the user may accept or deny the revisions at each step. The goal is to give the

user a concise set of revisions to choose from, yet also ones that are consistent with the original intent of the specification. Any revisions that are accepted then become certificates that, if upheld at runtime, will guarantee the mission success.

As an example, consider a collaborative scenario in which a mobile robot tasked with fetching parts in a factory setting, illustrated in Figure 3.1. In this scenario, the robot is required to continually visit the supply room and workstations, while avoiding any workstations that are occupied. It must be able to robustly avoid collisions with obstacles and appropriately react to the workstation as it is occupied or unoccupied. If this specification is applied to a vehicle with inertia, the robot's speed and deceleration will become a factor in synthesizing a controller that fulfills the the task. For instance, once a region is sensed as being occupied, the task could fail because the vehicle may not be able to stop by the time it reaches that region, violating the requirement *"avoid any occupied workstations"*. To accommodate the effect of inertia, we could recover if the user is given the environment assumption *"A workstation must not be occupied if the robot is within 1 meter of it,"* and it is accepted for inclusion as an additional assumption in the specification. Notice that, by giving the user the option to accept such assumptions, he/she is aware that the robot will succeed if that assumption is met. On the other hand, the robot may succeed if the assumption is not met, but there are no longer any formal guarantees for the task.

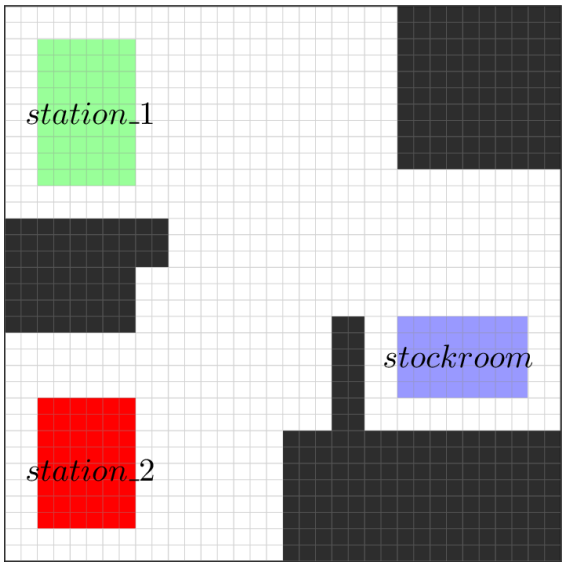


Figure 3.1: Factory resupply example scenario.

3.1.1 Related Work

Several researchers have focused attention to the problem of formal synthesis of controllers for physical systems. [10,64] have approached this problem from the standpoint of multi-layered synthesis, where certain parts of the control strategy are left open for an online planner to complete at runtime. Our approach is different in the sense that we seek controllers that guarantee the task under the dynamics at synthesis time, rather than computing a motion plan at runtime. Similar synthesis approaches (e.g. [57,58]) provide guarantees for nonlinear systems, but assume that the specification is realizable. This work is complementary in the sense that we strictly deal with the case of *unrealizability* due to the dynamics of the physical platform. Our approach, moreover, provides the user with a rich source of information regarding compatibility issues with the chosen platform. Specifically, we generate certificates that enable a user to consider the environment's behavior with respect to the mission and the dynamics of the

autonomous system.

Our approach for computing revisions is closely related to recent methods described in [5,34,55,56]. In [34], a method is devised for determining the cause of unrealizability for non-reactive tasks and providing specification recommendations to the user. In the reactive setting, [55] present a debugging method for unrealizable specifications based on templates (LTL formulas) mined from an environment counterstrategy. A counterstrategy captures the possible behaviors for the environment for which there are no safe system moves that allow it to fulfill its goals. The method in [5] generates specification templates automatically from the counterstrategy, yielding additional safety and liveness environment statements that remove all execution traces of the counterstrategy. The work of [56] apply the counterstrategy-based environment assumption mining technique to an early warning system in human-in-the-loop control systems, demonstrated in an autonomous driving scenario. By removing the behaviors present in the counterstrategy, the modified environment is restricted in such a way as to permit the system to realize its goals under the strengthened assumptions, but can sometimes lead to specifications that no longer match the user's intent.

The proposed approach differs from existing works in several ways. The closest work to ours, [5], adopt a general approach to specification revisions. Hence, the revisions generated do not hold preference in any one part of a counterstrategy over any other part, and to avoid placing unnecessary restrictions on either the environment or the behaviors of the system, it is up to the user to decipher which of the generated revisions are important to keep. Our approach, in contrast, proposes formulas that considers the discrete abstraction to guide

the creation of environment assumptions that render the specification realizable. The rationale is to propose a concise set of revisions for users to interpret rather than whole counterstrategies. This also aids in managing a large number of revisions.

Another difference from many existing works (e.g. [5,55]) is that we remove the burden for the user to choose templates and subsets of variables for revising specifications. Our algorithm does both automatically, allowing them to simply accept or reject the proposed certificates at each step of an iterative synthesis algorithm. We emphasize how we parse such formulas into statements that are simple to understand, making use of a graphical user interface where applicable.

Recent attention has focused on refining an existing abstraction as a means of rendering a specification realizable. For instance, [68] presents an approach to abstraction refinement in which the authors adopt a counterexample-guided abstraction procedure to iteratively re-partition the state space of a dynamical system, with the goal of satisfying the specification under the dynamical system. Another related work, [25], introduces a partitioning scheme that refines a discrete abstraction of a nonlinear system as a means for repairing unrealizable specifications that are *reactive* in nature. The main distinction is that we reason purely about an uncontrollable environment under a *given* discrete abstraction, so as to characterize the environment behaviors required to guarantee the task under this abstraction. To show the benefit of using our certificates to the task of abstraction refinement with respect to a reactive task, we adopt the state-space-partitioning abstraction refinement procedure of [25] as a case study in the robotics domain.

3.1.2 Outline

The remainder of this chapter is outlined as follows. In Section 3.2, we review relevant formal definitions and notation. We formally state the problem in Section 3.3 and present the approach for generating formulas and user feedback in Section 3.4. In Section 3.5, we present an approach for parsing the revisions as feedback to the user both as verbal statements and with the help of a graphical tool. Next, we demonstrate our approach for two case studies that employ two different types of discrete abstractions for a mobile robot are presented in Section 3.6. Lastly, we provide a summary in Section 3.7.

3.2 Preliminaries

3.2.1 Linear Temporal Logic

Linear temporal logic (LTL) formulas are defined over the set AP of atomic propositions in the recursive grammar:

$$\varphi ::= \mathit{true} \mid \pi \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \bigcirc\varphi \mid \varphi_1 \mathsf{U} \varphi_2$$

where π is an atomic proposition in AP. Respectively, $\wedge$ and $\neg$ are the Boolean operators "conjunction" and "negation", and $\bigcirc$ and U are the temporal operators "next" and "until". From these operators, the following operators are derived: "disjunction" $\vee$, "implication" $\Rightarrow$, "equivalence" $\Leftrightarrow$, "always" $\Box$, and "eventually" $\Diamond$.

AP consists of a set of *environment* propositions $\mathcal{X}$ describing the state of sensed environment (e.g. discretized values of a continuous-valued sensor) and

a set of *system action* propositions $\mathcal{Y}$ describing the discrete actions the system can take. The LTL formulas are evaluated over infinite sequences $\sigma = \sigma_0\sigma_1\sigma_2\ldots$ of truth assignments to the propositions in AP. σ is said to *satisfy* $\bigcirc\varphi$, $\Box\varphi$, or $\Diamond\varphi$ if φ holds true in the next position in the sequence, every position, or at some future position(s), respectively. We refer the reader to [88] for a complete definition of the syntax and semantics of LTL formulas.

3.2.2 Discrete Abstractions

Our model of the behavior of the physical system is a nonlinear differential equation

$$\dot{\xi}(t) = f(\xi(t), v(t)) \tag{3.1}$$

given by the function $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$, where $\xi(t)$ is the continuous state of the system and $v(t)$ the command input at time $t \in \mathbb{R}_{\geq 0}$. We impose the usual regularity assumptions on f that imply the existence and uniqueness of solutions of (3.1).

Abstractions of a Dynamical System

Given a bounded configuration space $\mathcal{W} \subset \mathbb{R}^n$, let $\mathcal{R} = \{R_1 \ldots R_p\}$ represent a set of regions (in general, not necessarily disjoint) whose closure covers $\mathcal{W}$, where the open sets $R_i \subseteq \mathcal{W}$. Wherever disjointness must hold, we will state so explicitly. The system (3.1) may be either open-loop, in which case the inputs v represent the low-level commands given to the system, or else closed-loop, in which case v are regarded as high-level commands such as a target region

that must eventually be reached under the action of some low-level (feedback) controller that drives the system from one point or region in the state space to another. We define discrete abstractions for both of these cases.

We adopt the motion encoding of [75] by introducing a *completion* proposition $\mathcal{X}_c \subseteq \mathcal{X}$, and let $\mathcal{X}_{nc} = \mathcal{X} \backslash \mathcal{X}_c$ denote those environment propositions not associated with completion (e.g. sensor propositions). Let $\pi_i \in \mathcal{X}_c$ denote a proposition that is `True` iff the system is in a certain configuration or region. Let $\pi_{aj} \in \mathcal{Y}$ denote a proposition that is `True` when the system is *activating* the jth motion command. We assume that all π_{aj} are mutually exclusive – the system can only activate one request at a time.

Definition 3.1 (***Discrete Abstraction).*** *We define a* discrete abstraction S_a *as the tuple* $(Q_a, V_a, \mathcal{X}_c, \mathcal{Y}, \gamma^a_{\mathcal{X}}, \gamma^a_{\mathcal{Y}}, \delta_a)$*, where:*

- Q_a *is the set of regions discretizing the system's configuration space;*
- V_a *is the set of discretized system locomotion commands;*
- $\mathcal{X}_c$ *and* $\mathcal{Y}$ *are, respectively, the configuration and command propositions (defined above);*
- $\gamma^a_{\mathcal{X}} : Q_a \rightarrow 2^{\mathcal{X}_c}$ *labels each region with the associated proposition in* $\mathcal{X}_c$ *that evaluates to* `True` *when the system is in the given region(s);*
- $\gamma^a_{\mathcal{Y}} : V_a \rightarrow \mathcal{Y}$ *labels each discrete command with the associated proposition in* $\mathcal{Y}$ *that evaluates to* `True` *when the system is activating the command;*
- $\delta_a : Q_a \times V_a \rightarrow 2^{Q_a}$ *is a nondeterministic transition relation defining a region* $\gamma^a_{\mathcal{X}}(q'_a) \in \mathcal{X}_c$ *once an action* $v_a \in V_a$ *is taken when in region* $\gamma^a_{\mathcal{X}}(q_a) \in \mathcal{X}_c$*, where* $q'_a \in \delta_a(q_a, v_a)$*.*

This abstraction yields an encoding that may be expressed by a set of LTL formulas [75]. Two possible semantics of the abstraction are explained in the case studies in Section 3.6.

Abstractions in the Absence of Dynamics

In the absence of dynamics, let the action propositions $\mathcal{Y}$ represent the workspace regions and $\pi_i \in \mathcal{Y}$ denote a region proposition that evaluates to `True` when the system is in $R_i \in \mathcal{R}$. We consider a topology model, an undirected connectivity graph describing those workspace regions that are accessible and adjacent to one another.

Definition 3.2 (***Topology Model).*** *We define a* topology model *as a formula φ_t^{top} over $\mathcal{Y}$ that encodes the allowed next regions given the current region, as follows:*

$$\varphi_t^{top} = \bigwedge_{\pi_i \in \mathcal{Y}} \square \left(\pi_i \implies \bigvee_{\substack{\pi_j \in \mathcal{Y}: \\ cl(R_i) \cap cl(R_j) \neq \varnothing}} \bigcirc \pi_j \right),$$

where $cl(\cdot)$ denotes the closure operation on a set. Note that we also enforce mutual exclusion of regions such that the physical system is only allowed to occupy one region at a time.

3.2.3 Controller Synthesis

We define a mission specification from which it is required to synthesize a controller for the system.

Definition 3.3 (***Mission Specifications).*** *The specifications we consider are LTL for-*

mulas of the form:

$$\varphi := \underbrace{(\varphi_i^e \wedge \varphi_t^e \wedge \varphi_g^e)}_{\varphi^e} \implies \underbrace{(\varphi_i^s \wedge \varphi_t^s \wedge \varphi_g^s)}_{\varphi^s}$$

The formulas φ_i^α, φ_t^α, and φ_g^α are defined over AP, where φ_i^α are formulas for the initial conditions, φ_t^α the allowed transitions (safety conditions) to be satisfied always, φ_g^α the goals (liveness conditions) to be satisfied infinitely often, and $\alpha = \{e, s\}$ (with e for 'environment' and s for 'system'). The liveness guarantees take the form $\bigwedge_{i \in I_\alpha} \square\diamondsuit(B_i^\alpha)$, where I_α is the index set of environment goals B_i^e, defined over $\mathcal{X} \cup \mathcal{Y} \cup \mathcal{X}' \cup \mathcal{Y}'$, or system goals B_i^s, defined over $\mathcal{X} \cup \mathcal{Y} \cup \mathcal{X}'$. $\mathcal{X}'$ and $\mathcal{Y}'$ are those propositions in $\mathcal{X}$ and $\mathcal{Y}$, respectively, prepended by the $\bigcirc$ operator.

Definition 3.4 (***Controller Finite State Machine and Execution).*** *A high-level controller is defined as a finite-state machine (FSM) $\mathcal{A} = (Q, Q_0, \mathcal{X}, \mathcal{Y}, \delta, \gamma_\mathcal{X}, \gamma_\mathcal{Y})$, where Q is the set of controller states, $Q_0 \subseteq Q$ is the set of initial controller states, $\mathcal{X}$ and $\mathcal{Y}$ are sets of propositions described above, $\delta : Q \times 2^\mathcal{X} \to Q$ is a state transition relation providing the next state $q' \in Q$ given the current state $q \in Q$ and the current value of the environment input $z \in 2^\mathcal{X}$, i.e. $q' = \delta(q, z)$, $\gamma_\mathcal{X} : Q \to 2^\mathcal{X}$ is a labelling function mapping controller states to the set of environment propositions evaluating to* `True` *for all transitions into that state, and $\gamma_\mathcal{Y} : Q \to 2^\mathcal{Y}$ is a labelling function mapping controller states to the set of action propositions evaluating to* `True` *in that state.*

Consider an infinite execution *σ of $\mathcal{A}$, where $\sigma = (\gamma_\mathcal{X}(q_0), \gamma_\mathcal{Y}(q_0))(\gamma_\mathcal{X}(q_1), \gamma_\mathcal{Y}(q_1)), \ldots$ for $q_0 \in Q_0$ and $q_i \in Q$, $i > 0$. At each step i in the execution, we say that the environment has made a move (occurring first) if there exists an assignment $\gamma_\mathcal{X}(q_i)$, and that the system has a move (occurring only after the environment has moved) if there exists an assignment $\gamma_\mathcal{Y}(q_i)$. A specification φ written over AP is deemed* realizable *if there exists a finite-state machine $\mathcal{A}$ such that every execution produced by $\mathcal{A}$ satisfies φ. That is, at every $i \geq 0$, there exists an assignment of system variables $\mathcal{Y}$ for*

all possible assignments of the environment variables $\mathcal{X}$ such that σ satisfies φ. If there exist some environment behaviors on $\mathcal{X}$ for which no such $\mathcal{A}$ can be found, then φ is unrealizable.

When combining a mission specification $\varphi^e \implies \varphi^s$ with the topological model, we obtain a *general formula*

$$\varphi := \varphi^e \implies (\varphi^s \wedge \varphi_t^{top}),$$

written over $AP = \mathcal{X} \cup \mathcal{Y}$.

When combining a mission specification with a discrete abstraction, we apply the proposition mapping $\mathcal{X}_{nc} \leftarrow \mathcal{X}$, $\mathcal{X}_c \leftarrow \mathcal{Y}$ and define $\mathcal{Y} = \{\pi_{aj}\}_j$, where each π_{aj} is a robot motion command. We then obtain a *platform-specific formula*

$$\varphi^{abs} := (\varphi^e \wedge \varphi_{t,g}^{e,a}) \implies (\varphi^s \wedge \varphi_t^{s,a}),$$

written over $\mathcal{X}_c \cup \mathcal{X}_{nc} \cup \mathcal{Y}$ in which $\varphi_{t,g}^{e,a}$ consists of environment liveness and safety formulas and $\varphi_t^{s,a}$ consists of system safety formulas, both over $\mathcal{X}_c \cup \mathcal{Y}$, where the superscript a stands for 'abstraction'. Note that φ_t^{top} no longer appears in φ^{abs}. Details on this encoding for specific abstractions presented in Section 3.6 may be found in [19,25,75].

If φ^{abs} is realizable, then a finite-state machine $\mathcal{A}$ is synthesized by solving a two-player game played between the environment and the system, using a GR(1) synthesis algorithm described in [11].

In the case that φ^{abs} is not realizable, we can synthesize an *environment counterstrategy*: a state machine that captures the possible behaviors for the environment preventing the system from fulfilling its goals.

Definition 3.5 (***Environment Counterstrategies).*** *We define an* environment counterstrategy *as a finite-state machine* $\mathcal{A}_c = (Q_c, Q_{c0}, \mathcal{X}, \mathcal{Y}, \delta_c, \gamma^c_{\mathcal{X}}, \gamma^c_{\mathcal{Y}})$*, where*

- Q_c *is the set of counterstrategy states;*
- $Q_{c0} \subseteq Q_c$ *is the set of initial counterstrategy states;*
- $\mathcal{X}$, $\mathcal{Y}$ *are sets of propositions in AP;*
- $\delta_c : Q_c \times 2^{\mathcal{Y}} \to 2^{Q_c}$ *is a nondeterministic transition relation returning the set of counterstrategy states at the next position in the sequence given the current state and the current valuations of system commands in* $\mathcal{Y}$*;*
- $\gamma^c_{\mathcal{Y}} : Q_c \to 2^{\mathcal{Y}}$ *is a labelling function mapping counterstrategy states to the set of action propositions that are* `True` *for all transitions into that state, and;*
- $\gamma^c_{\mathcal{X}} : Q_c \to 2^{\mathcal{X}}$ *is a labelling function mapping counterstrategy states to the set of environment propositions that are* `True` *in that state.*

For notational convenience, we define $\delta^c(q) = \{q' \in Q_c | \exists y_e \in \mathcal{Y} : q' \in \delta_c(q, y_e)\}$ as the projection of $\delta_c(\cdot, \cdot)$ onto the set Q_c, and $\delta^{c^{-1}}(q') = \{q \in Q_c \mid q' \in \delta^c(q)\}$ as the inverse transformation relation mapping counterstrategy states to a set of predecessors.

We now briefly outline how such counterstrategies are synthesized, where the reader is referred to [49] for technical details. Synthesis involves first performing a fixed point computation on a representation of the specification to find a Boolean formula of the winning positions for the environment. From these winning positions, a particular counterstrategy $\mathcal{A}_c$ is extracted. Here, we use the term *positions* to denote assignments over $\mathcal{X} \cup \mathcal{Y} \cup \mathcal{X}' \cup \mathcal{Y}'$ for both the environment and system. We may express the environment's winning positions

as

$$WP_{env} = \mu\mathcal{V}_1.\bigvee_{i_s \in I_s} \nu\mathcal{V}_2.\bigwedge_{i_e \in I_e} \mu\mathcal{V}_3.\left(\neg B^s_{i_s} \vee \mathsf{Pre}\mathcal{V}_1\right) \wedge \mathsf{Pre}\mathcal{V}_2 \wedge \left(B^e_{i_e} \vee \mathsf{Pre}\mathcal{V}_3\right),$$

where μ and ν are, respectively, the least and greatest fixpoint operators, and where $\mathcal{V}_1$ and $\mathcal{V}_3$ are initially `True`, and $\mathcal{V}_2$ is initially `False` before the fixed point computation begins. $\mathsf{Pre}\mathcal{V}$ is an enforceable predecessor operator that produces, for a set of positions $\mathcal{V}$, a set of predecessor positions such that, there exists an environment assignment satisfying the environment safety formulas φ^e_t and $\varphi^{e,a}_t$, for all possible action assignments satisfying the system safety formulas φ^s_t and $\varphi^{s,a}_t$.

3.3 Problem Formulation

Assume the original mission specification φ under the assumption of a topological model φ^{top}_t is *realizable*, where the formulas φ^e and φ^s are, respectively, environment assumptions and system guarantees defined by the user. Additionally, we require that φ^e is not falsified by system behaviors satisfying φ^s (i.e. no trivial behaviors). The goal is to synthesize controllers for such specifications. Given the discrete abstraction and the formulas φ^e and φ^s, we seek a controller for the platform-specific specification φ^{abs}.

If φ is realizable and φ^{abs} is *unrealizable*, then, according to Definition 3.4, there exists some environment behavior in $\mathcal{X}_{nc}$ admissible by φ^e such that, if behaviors in $\mathcal{X}_c$ satisfy $\varphi^{s,e}_{t,g}$, then no system behaviors satisfy $\varphi^s \wedge \varphi^{s,a}_t$.

In the case that φ^{abs} is *unrealizable*, the goal of this chapter is to generate a set of revisions (LTL formulas) that, upon conjunction with the platform-specific

formula φ^{abs}, render it realizable. Specifically:

Problem 3.1 (***Revision Generation and User Feedback).*** *Given φ realizable, φ^{abs} unrealizable, automatically derive a set of formulas for the environment and the system such that*

$$\varphi^{mod} := (\varphi^{e} \wedge \varphi^{e,a}_{t,g} \wedge \psi^{e}_{g} \wedge \psi^{e}_{t}) \implies (\varphi^{s} \wedge \varphi^{s,a}_{t} \wedge \psi^{s}_{t}) \tag{3.2}$$

is realizable and both $\varphi^{e} \wedge \varphi^{e,a}_{t,g} \wedge \psi^{e}_{g} \wedge \psi^{e}_{t}$ and $\varphi^{s} \wedge \varphi^{s,a}_{t} \wedge \psi^{s}_{t}$ are satisfiable. For any such revisions, provide the user with a suggested set of runtime certificates: a set of human-readable statements consisting of safety assumptions and guarantees, resp. ψ^{e}_{t} and ψ^{s}_{t}, and liveness assumptions ψ^{e}_{g}.

To illustrate the problem, consider the following example.

Example 3.1. *Consider again the factory setting of Figure 3.1. We begin by writing the specification in terms of a topology model. Under such a model, the regions of the workspace to be visited are encoded as actions, hence stockroom, station_1 and station_2 belong to $\mathcal{Y}$, while s1_occupied and s2_occupied (indicating when the respective region is occupied) are sensors belonging to $\mathcal{X}$. The robot is required to visit the stockroom and the two workstations (system liveness) but avoid visiting those that are occupied (system safety). If the robot is within a workstation, the environment is required to keep that station unoccupied (environment safety). Also, the workstations are required to be infinitely often unoccupied (environment liveness).*

The general specification φ is composed of the following formulas:

$$\Box\Diamond\ stockroom \wedge \Box\Diamond\ station\_1 \wedge \Box\Diamond\ station\_2 \qquad \triangleleft\ sys\ liveness$$

$$\Box\Diamond \neg s1\_occupied \wedge \Box\Diamond \neg s2\_occupied \qquad \triangleleft\ env\ liveness$$

$$\Box(\bigcirc\ s1\_occupied \implies \bigcirc \neg station\_1) \qquad \triangleleft\ sys\ safety$$

$$\Box(\bigcirc\ s2\_occupied \implies \bigcirc \neg station\_2) \qquad \triangleleft\ sys\ safety$$

$$\Box(station\_1 \implies \bigcirc \neg s1\_occupied) \qquad \triangleleft\ env\ safety$$

$$\Box(station\_2 \implies \bigcirc \neg s2\_occupied) \qquad \triangleleft\ env\ safety$$

$$\texttt{True} \qquad \triangleleft\ sys\ init$$

$$\texttt{True} \qquad \triangleleft\ env\ init$$

and the following topology model:

$$\begin{aligned}\phi_t^{top} = &\Box\,(stockroom \implies \bigcirc\, factory\_floor \bigcirc\, stockroom) \wedge \\ &\Box\,(station\_1 \implies \bigcirc\, factory\_floor \vee \bigcirc\, station\_1) \wedge \\ &\Box\,(station\_2 \implies \bigcirc\, factory\_floor \vee \bigcirc\, station\_2) \wedge \\ &\Box\,(factory\_floor \implies \bigcirc\, stockroom \vee \bigcirc\, station\_1 \vee \\ &\qquad \bigcirc\, station\_2 \vee \bigcirc\, factory\_floor)\,.\end{aligned}$$

where $factory\_floor$ corresponds to the unlabeled white space in Figure 3.1. The specification φ is realizable.

Now suppose we are given a fully-actuated planar robot governed by inertia, described by the system

$$\ddot{x} = u \qquad \ddot{y} = v, \tag{3.3}$$

where $(u, v) \in U$ are robot commands and $(x, y) \in \mathbb{R}^2$ are the Cartesian robot coordinates. We derive an abstraction S_a of these dynamics in the configuration space $(x, y, \dot{x}, \dot{y})^T \in$

$\mathcal{W}$ and obtain the formula φ^{abs}. With S_a, we want to synthesize a realizable controller for φ^{abs} that satisfies the task given an abstraction of these dynamics.

The specification may be unrealizable for a number of different reasons. One cause is *deadlock*, where the environment can force the system into certain states that have no legal transitions. Suppose that the robot has inertia, which is encoded in the abstraction as requiring two regions before it is able to decelerate to a stop. If $s1\_occupied$ turns from `False` to `True` when the robot is within two grid cells of $station\_1$, it will be unable to avoid a collision. Note that, as this behavior stems from the robot's physics, this behavior occurs in the platform-specific specification but not in the general specification.

Another cause of unrealizability is due to *livelock*, in which the system is prevented from reaching its goals as a result of an infinite sequence of environment inputs. For instance, suppose the robot is approaching $station\_1$ and the environment toggles the value of $s1\_occupied$ infinitely often. If the switching is fast enough, the robot may be able to change its heading, but unable to move forward toward the workstation. The behavior does not appear in the general formula because the topology graph always allows the robot to either remain in place or transit to an adjacent region once the environment has made a move.

3.4 Revising Unrealizable Specifications via Counterstrategies

In this section, we formalize our solution strategy for Problem 3.1. The goal is to generate revisions that, if possible, repair unrealizable specifications where the unrealizability is a consequence of a discrete abstraction included in the specification. We adopt an iterative approach that takes a specification φ and

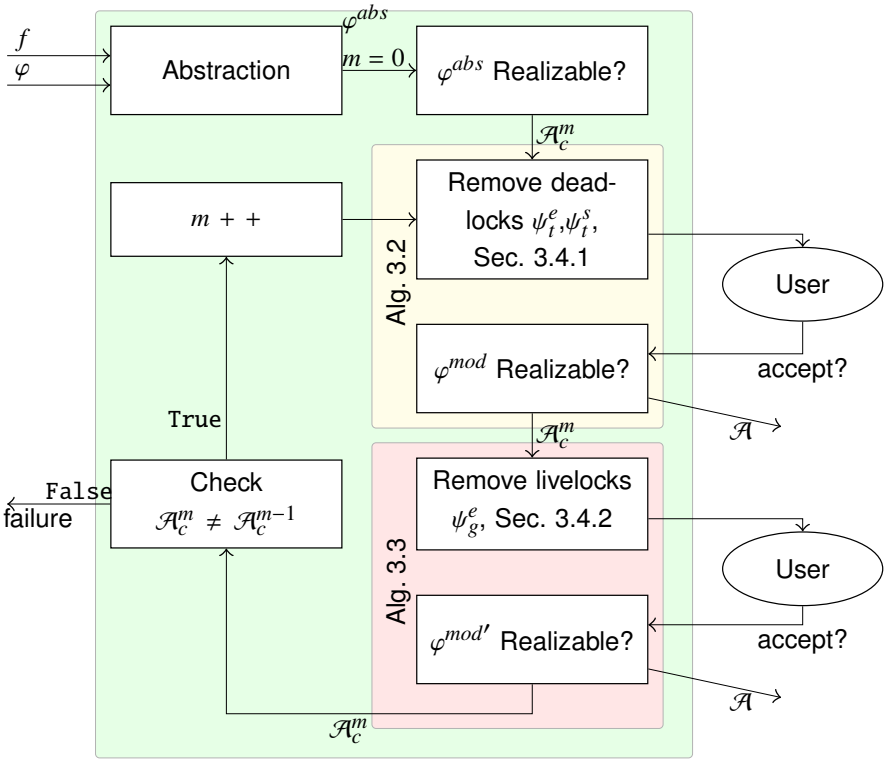


Figure 3.2: Overview of the procedure for finding runtime certificates and controller synthesis.

dynamical system f, and interacts with the user at various stages of the process, as illustrated in Figure 4.5. If successful, the approach outputs a finite state machine $\mathcal{A}$ and the user is exposed to the revisions that have been added to the specification. At each step, the user may choose to accept the revisions or else supply their own handwritten revisions. The interactive nature of the algorithm allows the specification designer to choose revisions that are consistent with his or her intent.

The first step is to create an abstraction of the specification (Abstraction in Figure 4.5), either via a gridding procedure [19] or via the constructive procedure giving rise to partitions based on reachability analysis [20]. In the latter case, Abstraction contains a call to synthesize a finite-state machine from φ,

upon which atomic controllers may be constructed. If the specification φ^{abs} or the modified specification φ^{mod} is unrealizable, various stages of the revisions approach are invoked as necessary; the complete process is discussed in this section. The Realizable blocks take as input a specification and returns a controller FSM $\mathcal{A}$ if realizable; otherwise, a counterstrategy $\mathcal{A}_c$ is returned. $\mathcal{A}_c$ will differ depending on whether it is synthesized from a deadlock-modified specification or livelock-modified specification. Note that, if the specification is unrealizable and the counterstrategy is the same between iterations of the while loop, this means that no revisions have been found that meet the user's criteria or do not falsify the specification. In this case, the algorithm terminates with an *unrealizable* output.

Note that there are many potential counterstrategies for a given unrealizable specification, with each one being assembled from a *game structure* [11] and contain a subset of behaviors of the game structure. When extracting revisions to the specification, we reason on counterstrategies rather than on game structures for two reasons. First, game structures have at least as many behaviors as a counterstrategy, so an approach that extracts revisions on a game structure will produce at least as many revisions as one applied to a counterstrategy of that game structure. Having fewer revisions in general reduces the conservatism and lessens the number of assumptions that are fed to the user. Second, there are many possible counterstategies for a given game structure to draw from. Using existing counterstrategy-synthesis tools (e.g. [12,32]), these can be readily customized to suit a particular designer's needs. For instance, a counterstrategy could be extracted that minimizes some objective function (e.g. minimizes distance to a goal) or that gives preference to certain states over others (e.g. wide passageways over narrow ones). Thus, our aim is to introduce an approach

that generates a small number of revisions, and may be customized to a user's design intent.

We introduce the following example to illustrate the major concepts discussed in the remainder of this section.

Example 3.2. *Consider the workspace shown in Figure 3.3(a). Given $\mathcal{X} = \{sen\}$ where sen is the sensor input and $\mathcal{Y} = \{r1, r2\}$, we write a specification φ requiring the robot to visit $r2$ (lower-left gray region) when sen is* `False`*, but avoid $r2$ when sen is* `True`*. Formally:*

$$\begin{aligned}
&\square\diamondsuit\, r2 && \triangleleft\, \varphi_g^s\\
&\square\diamondsuit\, \neg sen && \triangleleft\, \varphi_g^e\\
&\square(\bigcirc\, sen \implies \bigcirc\, \neg r2) && \triangleleft\, \varphi_t^s\\
&\square(r2 \implies \bigcirc\, \neg sen) && \triangleleft\, \varphi_t^e\\
&\texttt{True} && \triangleleft\, \varphi_i^s\\
&\texttt{True} && \triangleleft\, \varphi_i^e
\end{aligned}$$

The initial conditions are denoted `True` *to signify that an execution can begin with any safe initial robot configuration, action, and environment settings. The controller satisfying this specification is given in Figure 3.3(b).*

We are now given an abstraction, automatically derived using the procedure in [75], in which we have redefined the complete set of environment variables to be $\mathcal{X} = \mathcal{X}_c \cup \mathcal{X}_{nc}$, where we map the set of environment propositions in φ to the set $\mathcal{X}_{nc}$ (in this case $\mathcal{X}_{nc} = \{sen\}$). The set $\mathcal{X}_c = \{x1, \ldots, x16\}$ is an encoding of the set of 2-D robot configurations, and we replace the set of robot actions $\mathcal{Y}$ with the robot's four cardinal directions of motion. An excerpt of two of the conjuncts in φ^{abs} for which the robot's current position

is x7 is as follows:

$$\square((x7 \wedge W) \implies (\bigcirc x6 \vee \bigcirc x7)) \wedge \square((x7 \wedge S) \implies (\bigcirc x11 \vee \bigcirc x7)) \triangleleft \varphi_t^s,$$

$$\square\diamond((x7 \wedge W) \implies \bigcirc x6) \wedge \square\diamond((x7 \wedge S) \implies \bigcirc x11) \qquad \triangleleft \varphi_g^e.$$

Additionally, φ^{abs} includes conjuncts in φ_t^s and φ_t^e that encode, respectively, mutual exclusion of action and completion propositions (no two actions or completions may occur at the same time). The complete discrete abstraction S_a appears as arrows in Figure 3.3(a). A new specification φ^{abs} is derived, where:

$$\begin{aligned}
&\square\diamond(x9 \vee x13) && \triangleleft \varphi_g^s \\
&\square\diamond \neg sen && \triangleleft \varphi_g^e \\
&\square(\bigcirc sen \implies \bigcirc\neg(x9 \vee x13)) && \triangleleft \varphi_t^s \\
&\square((x9 \vee x13) \implies \bigcirc\neg sen) && \triangleleft \varphi_t^e \\
&\texttt{True} && \triangleleft \varphi_i^s \\
&\texttt{True} && \triangleleft \varphi_i^e
\end{aligned}$$

3.4.1 Preventing Deadlock

In preventing deadlock, we introduce a scheme to process a counterstrategy and extract a set of environment assumptions that remove deadlock behaviors. Consider a counterstrategy $\mathcal{A}^c$ whose deadlock states are collected in $Q_{dead} = \{q \in Q_c \mid \delta^c(q) = \varnothing\}$, and let

$$B_{robot}(q) = \bigwedge_{\pi \in \gamma_{\mathcal{Y}}^c(q) \cup (\mathcal{X}_c \cap \gamma_{\mathcal{X}}^c(q))} \pi \wedge \bigwedge_{\pi \in (\mathcal{Y} \cup \mathcal{X}_c) \setminus (\gamma_{\mathcal{Y}}^c(q) \cup \gamma_{\mathcal{X}}^c(q))} \neg\pi,$$

$$B_{env}(q) = \bigwedge_{\pi \in \mathcal{X}_{nc} \cap \gamma_{\mathcal{X}}^c(q)} \pi \wedge \bigwedge_{\pi \in \mathcal{X}_{nc} \setminus \gamma_{\mathcal{X}}^c(q)} \neg\pi.$$

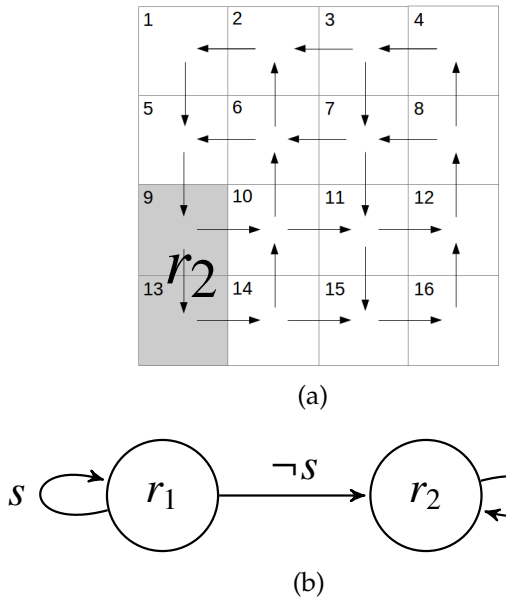


Figure 3.3: 2-D example. (a) shows the workspace map and grid whose cells are labeled with the configuration variable. The white grid cells denote r_1, while the gray denote r_2. (b) shows the synthesized controller for φ.

In words, $B_{robot}(q)$ denotes a Boolean formula for the truth-values of the command and configuration propositions $\mathcal{Y} \cup \mathcal{X}_c$ at state q and $B_{env}(q)$ denotes a Boolean formula for the truth-values of the subset of environment input propositions $\mathcal{X}_{nc}$ at state q.

Removing Deadlock

As in [5], any pair of states $q^i, q^j \in Q_c$ in a counterstrategy satisfying $q^j \in \delta^{c^{-1}}(q^i)$ can be expressed with the formula

$$\bigvee_{q^j \in \delta^{c^{-1}}(q^i)} \Diamond \left(B_{robot}(q^j) \wedge \bigcirc B_{env}(q^i) \right). \tag{3.4}$$

For a particular q^i, the formula characterizes the environment's behavior at this state in the counterstrategy and the robot's configuration at the state immedi-

ately prior. If there exists an execution that eventually reaches a deadlock state $q^i_{dead} \in Q_{dead}$, we may use this statement to remove the environment behaviors in the counterstrategy causing deadlock. The negation of (3.4) leads to the following formula, which is suggested as an additional assumption:

$$\bigwedge_{q^j \in \delta^{c^{-1}}(q^i_{dead})} \square \left(B_{robot}(q^j) \implies \bigcirc \neg B_{env}(q^i_{dead}) \right). \tag{3.5}$$

Before conjuncting each computed formula with ψ^e_t, a check is made to determine if it falsifies the left-hand side of φ^{mod}, i.e. if (3.5) is assigned to $\varphi^{e,a}_t$, then $\varphi^e \wedge \varphi^{e,a}_t \wedge \psi^e_t = \texttt{False}$. If this is the case, the formula is discarded and it is not included as a conjunct in ψ^e_t. The deadlock state index i is then incremented and the process repeats with a new suggested assumption.

Preventing Unintended Behaviors

When the environment assumptions of (3.5) are added to the specification, the system may exhibit behaviors that are noticeably different from the system's behaviors from the original specification synthesized under a topology graph. For instance, consider Example 3.2. Under the original (realizable) specification φ, if the robot is in the white region in Figure 3.3(a) and the door is closed, the FSM of Figure 3.3(b) reveals that the system remains within the white region until the door opens. Now consider the platform-specific specification φ^{abs} abstracted under the shown workspace decomposition. After applying assumptions to prevent the door from closing when the system is in cell $x5$ (where the system is unable to avoid entering r_2 in the next step), the system will not avoid moving toward the door when within the white region and the door is closed, clearly different behavior than that in Figure 3.3(b).

To reconcile these differences, our approach enforces that the system react conservatively to the newly-added environment revision by treating the state $q^j \in \delta^{c^{-1}}(q^i_{dead})$ (in the antecedent of (3.5)) as if it were a deadlock state. Thus, when the physical system is at a configuration previous to the deadlocked configuration and when the added assumptions on the environment prevent it from behaving in a certain way in the next state, it will be forbidden from entering the configuration prior to deadlock when these conditions hold in the current state. In other words, such conditions will prevent the system from approaching the door when it is closed. On the other hand, when the door is open, ψ^e_t prevents it from closing again once the system has made its move.

Formally, we disallow the behavior

$$\bigvee_{q^j \in \delta^{c^{-1}}(q^i_{dead})} \Diamond(\bigcirc B_{env}(q^i_{dead}) \wedge \bigcirc B_{robot}(q^j))$$

by introducing an additional revision ψ^s_t:

$$\bigwedge_{q^j \in \delta^{c^{-1}}(q^i_{dead})} \Box\left(\bigcirc B_{env}(q^i_{dead}) \implies \bigcirc \neg B_{robot}(q^j)\right). \tag{3.6}$$

Such a revision places a safety restriction on the system, preventing it from entering a *neighboring* state to a deadlock state whenever the environment is set to the same value for which deadlock occurs. Doing this produces a specification that makes the system's behavior conservative; we are limiting the conditions under which the system may enter a neighboring state, when in fact the system is not in any true danger of violating the original safety guarantees in φ until it reaches r_2. Nonetheless, if the specification is realizable, the system will be able to react to the environment as long as the actions/configurations are not included in those specified in $\bigwedge_{q^j \in \delta^{c^{-1}}(q^i_{dead})} B_{robot}(q^j)$.

Aggregated Deadlock Removal via Backward Reachability

If the modified formula is determined to be unrealizable and new deadlock states are found at a state $q^j \in \delta^{c^{-1}}(q^i_{dead})$, then we once again return to the original set of circumstances specified in Problem 3.1. We repeat the process in this section for as many times as required to eliminate deadlock states or until the resulting specification is unrealizable. This, however, has the drawback that synthesis of a counterstrategy may have to be repeated several times. We adopt a more direct approach that reduces the number of computations, and has the added benefit of producing user-generated statements that are simple to interpret (see Section 3.5).

To avoid repeated synthesis of counterstrategies, we apply the assumption and guarantee revisions explained above to entire *subtraces* of a single counterstrategy (a finite word of an execution trace for the counterstrategy). To do this, we identify states for which there is no safe command to be taken such that there exists a subtrace that eventually visits states in $Q_c \backslash Q_{dead}$. The search for deadlock revisions then reduces to a graph search on the counterstrategy, as summarized in Algorithm 3.1. The algorithm builds up a set of *deadlock-committed* states Q_{commit} by querying the abstraction and adding, via a breadth-first search (BFS in line 4), predecessor counterstrategy states from deadlock Q_{dead} for which all system commands lead to states in Q_{commit}. As such, the procedure BFS amounts to a backward reachability operation.

For generating revisions and providing user feedback, we also maintain a mapping $Q_{reach} : Q_{commit} \rightarrow 2^{Q_{dead}}$ of deadlock states reachable from each $q \in Q_{commit}$ obtained from the abstraction S_a. The search continues until a fixed point of states is reached where no additional deadlock-committed states can

be found, at which point BFS returns a tuple containing Q_{commit} and Q_{reach}. The precise condition under which the search terminates is when a $q \in Q_c$ is found such that:

$$\exists q' \in \delta^c(q) : q' \notin Q_{commit}.$$

Here, Q_{commit} plays the role of Q_{dead}. We therefore replace Q_{dead} in the safety revisions (3.5) and (3.6) with Q_{commit}. To be precise, we replace (3.5) and (3.6) with, respectively:

$$\bigwedge_{q^j \in Q^i_{commit}} \square \left(B_{robot}(q^j) \implies \bigwedge_{q^k \in Q_{reach}(q^i_{commit})} \bigcirc \neg B_{env}(q^k) \right) \tag{3.7}$$

$$\bigwedge_{q^j \in Q^i_{commit}} \square \left(\bigwedge_{q^k \in Q_{reach}(q^i_{commit})} \left(\bigcirc B_{env}(q^k) \implies \bigcirc \neg B_{robot}(q^j) \right) \right), \tag{3.8}$$

for each $q^i_{commit} \in Q_{commit}$.

Algorithm 3.2 contains a procedure for finding revisions that target deadlocks. The first step in the algorithm is to compute environment and system transition subformulas ψ^e_t and ψ^s_t (using the approach described in Section 3.4.1) that prevent transitions to states in the counterstrategy from which the system has no safe transitions (deadlock). If these revisions falsify the environment and system, they are removed. The second step is to provide feedback to the user. Depending on the user's response, the revisions are either applied or discarded. If accepted, they become runtime certificates, as discussed in Section 3.5.

The following proposition is immediate considering the fact that Algorithm 3.1 traverses a particular counterstrategy $\mathcal{A}_c$ via backward reachability using the transition system S_a.

Proposition 3.1. *Algorithm 3.1 is sound and complete with respect to S_a.*

Algorithm 3.1: Computing deadlock-committed states.

```
1: procedure COMMITSTATES(Q_dead)
2:     Initialize Q_new, Q_commit to Q_dead.
3:     while Q_new ≠ ∅ do
4:         Q_new ← BFS(A_c, Q_commit)
5:         Q_commit ← Q_commit ∪ Q_new
6:         for q ∈ Q_new do
7:             Q_reach(q) ← δ^c(q)          ▷ Create a graph of reachable states
8:         end for
9:     end while
10:    return Q_commit, Q_reach
11: end procedure
```

Soundness with respect to S_a implies that Q_{reach} does not contain states for which there exists no sequence of actions that may eventually lead to Q_{dead}. *Completeness with respect to* S_a implies that, from each state in Q_{reach}, there exists a sequence of actions that may eventually reach Q_{dead} and Algorithm 3.1 will always terminate with such a Q_{reach}, if it exists.

Example 3.3. *Returning to Example 3.2, suppose we obtain a counterstrategy containing the states* $q_0, \ldots, q_3$*, as pictured in Figure 3.4(a) and Figure 3.4(b), starting in cell* $x7$ *with sen* = `False`*. One of the possible executions in this counterstrategy eventually leads the robot to cell* $x5$ *with the sensor sen* = `True` *as shown in Figure 3.4(b). In this execution, the sensor sen remains* `False` *until the robot enters* $x5$*, at which point a transition in* φ_t^s *is violated. Hence* q_3 *is a deadlock state. The formula* $\Diamond(\bigcirc sen \wedge \neg x6 \wedge W)$[1] *(in words, "eventually, the system will be in cell 6 and activating go West, with sen* `True` *in the following time step") is extracted by evaluating* $\Diamond(B_{robot}(q_2) \wedge \bigcirc B_{env}(q_3))$.

[1] We only make the `True` action explicit (W in this case), since mutual exclusion disallows the other actions from being activated at the same time.

Algorithm 3.2: Synthesizing deadlock revisions for an realizable specification φ^{abs}.

procedure SYNTHDEADLOCKREVISIONS(φ^{abs},$\mathcal{A}_c$,$\mathcal{R}$)
  $Q_{commit} \leftarrow$ commitStates($\mathcal{A}_c$)
  for all $q^i_{commit} \in Q_{commit}$ **do** ▷ *Eliminate deadlocks*
    $\psi^e_{t,cand}, \psi^e_t \leftarrow$ Eq. (3.7)
    $\psi^s_{t,cand}, \psi^s_t \leftarrow$ Eq. (3.8)
    if $\neg(\varphi^e \wedge \varphi^{e,a}_{t,g} \wedge \psi^e_g \wedge \psi^e_t \wedge \psi^e_{t,cand})$ or $\neg(\varphi^s \wedge \varphi^{s,a}_t \wedge \psi^s_t \wedge \psi^s_{t,cand})$ **then**
      $\psi^e_t \leftarrow \psi^e_t \backslash \psi^e_{t,cand}$
      $\psi^s_t \leftarrow \psi^s_t \backslash \psi^s_{t,cand}$
    end if
  end for
  for all $R_i \in \mathcal{R}$ **do** ▷ *User feedback*
    $(q^\star_i, q^\star_{dead}) \leftarrow$ Eq. (3.22)
    $dist_i \leftarrow |\gamma^c_X(q^\star) - \gamma^c_X(q^\star_{dead})|$
    print $(R_i, dist_i, B_{env}(q^\star_{dead,i}))$
  end for
  if user accepts any ψ^e_t, ψ^s_t **then** ▷ *Add to the specification and attempt to synthesize a new counterstrategy*
    $\varphi^{mod} \leftarrow$ Eq. (3.2)
    $(realiz, \mathcal{A}^m_c, WP_{env}) \leftarrow$ ctrStrategy(φ^{mod})
  end if
  return $realiz$,$\mathcal{A}_c$,WP_{env},φ^{mod}
end procedure

The complement of this formula, $\square((\neg x6 \wedge W) \implies \bigcirc \neg sen)$*, is added as an additional environment assumption. This assumption negates the behaviors in the counterstrategy for that particular deadlock state. Being that there is only one deadlock state, we add no further assumptions. Upon adding this revision to the environment assumptions, we determine that the modified specification is realizable.*

Notice that the controller synthesized based on the specification above produces executions that satisfy the specification but the system now assumes that the environment will always turn sen `False` *whenever it reaches* $x5$*. In the example, consider the behavior when the robot starts at* $x7$ *with sen* `True`*. The execution of the robot in this case is as shown in Figure 3.4(c). In this execution, sen remains* `True` *and, as the robot moves toward* $r2$*, the environment eventually must set sen to* `False` *to be consistent with the added assumption. When outside of* $x5$*, the robot follows the same sequence of moves regardless of the environment. Note that the controller for the original, realizable specification* φ *(Figure 3.3(b)) does not exhibit this behavior because there is no imposition on how the environment must behave based on the robot's configuration. In that case, if sen is* `True`*, the robot waits in* $r1$ *until sen becomes* `False`*.*

We compute a set of four deadlock-commit states $Q_{commit} = \{q'_1, q'_2, q'_3, q'_4\}$ *corresponding to the cells* $\{x5, x1, x2, x6\}$*. We obtain the following* ψ_t^e *formulas:*

$$\square((x5 \wedge S) \implies \bigcirc \neg sen) \tag{3.9}$$

$$\square((x1 \wedge S) \implies \bigcirc \neg sen) \tag{3.10}$$

$$\square((x2 \wedge W) \implies \bigcirc \neg sen) \tag{3.11}$$

$$\square((x6 \wedge N) \implies \bigcirc \neg sen), \tag{3.12}$$

and the following ψ_t^s *formulas:*

$$\square(\bigcirc sen \implies \bigcirc \neg(x5 \wedge S)) \tag{3.13}$$

$$\square(\bigcirc sen \implies \bigcirc \neg(x1 \wedge S)) \tag{3.14}$$

$$\square(\bigcirc sen \implies \bigcirc \neg(x2 \wedge W)) \tag{3.15}$$

$$\square(\bigcirc sen \implies \bigcirc \neg(x6 \wedge N)). \tag{3.16}$$

With these revisions added to ψ_t^e *and* ψ_t^s *(highlighted orange in Figure 3.5, the modified specification eliminates the deadlock states present in the original counterstrategy. Ob-*

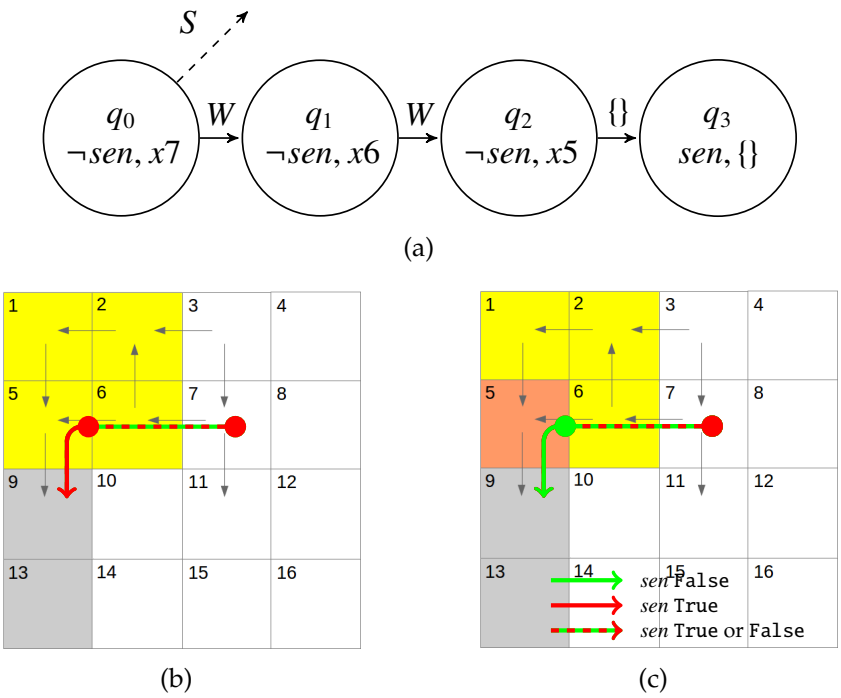


Figure 3.4: (a) shows a partial counterstrategy for Example 3.3 leading to deadlock. (b) shows a corresponding robot trajectory leading to deadlock. The cells shaded yellow indicate configurations in which there are no sequence of commands that avoid reaching r_2 eventually. (c) shows the result of a synthesized controller where deadlock is removed, but where the strategy *expects* the environment to set *sen* to `False` once the robot enters cell $x5$. The numbering of the cells correspond to the state labels, omitting the "x".

serve that (3.9) *alone will eliminate deadlock at cell* $x5$*; however,* (3.9) *coupled with the system safety formulas* (3.13) *will introduce another deadlock at* $x1$ *and* $x6$*, and so on. Hence,* (3.9) – (3.16) *are necessary when both environment and system safety formulas* (3.7) *and* (3.8) *are added at each state in* Q_{commit}*. Additionally, note that none of the revisions falsify the environment.*

Upon synthesis, we find that a counterstrategy synthesized from this modified specification does not contain deadlock states. In the next section, we dis-

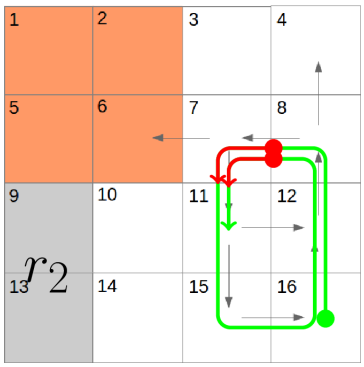


Figure 3.5: Map showing configurations for which the revisions ψ_t^e and ψ_t^s from Example 3.3 apply; a counterstrategy execution trace, as explained in Section 3.4.2. The green part of the path denotes where *sen* = `False` and the red denotes where *sen* = `True`.

cuss an approach to render the specification realizable through an elimination of livelock behaviors.

3.4.2 Preventing Livelock

The environment may be able to win the two-player game through livelock: moves for the environment that force the system to cycle indefinitely through a sequence of states, keeping the system away from one of its goals. Consider the behavior of the system when the above ψ_t^e and ψ_t^s formulas (3.9)–(3.16) are introduced as revisions. Starting at $x16$, the behavior shown in Figure 3.5 is possible. In this execution (shown in Figure 3.5), the system eventually cycles indefinitely between six cells in the workspace. Whenever the robot visits the cell $x7$, the environment activates *sen*, forcing the robot to move S to avoid violating the safety guarantee revision in (3.16). The environment is then able to satisfy its liveness goal ($\square\diamond(\neg sen)$), while preventing the system from achieving its goal of reaching r_2.

Once we obtain a counterstrategy free of deadlock states, our approach generates environment assumptions that remove the counterstrategy executions that exhibit livelock. The idea is to selectively identifies states in the counterstrategy for which the system still has winning actions to take. Our approach then exploits states with this property to prevent the environment from always making such assignments at these counterstrategy states. We do so by applying liveness assumptions that remove the environment's ability to remain in these states forever. Hence, there exists some finite time in the execution where the system is allowed to take these actions.

Take any $q \in Q_c$. Let $V_{nc} : Q_c \to 2^{\mathcal{X}_{nc}}$ be a function mapping counterstrategy states to the set of all non-completion environment proposition assignments at the current step in the execution that satisfy the environment's safety formula φ_t^e, with respect to the proposition assignments at the *previous* step $\gamma_{\mathcal{X}}^c(q) \cup \gamma_{\mathcal{Y}}^c(q)$. Now, let

$$B'_{nc}(q) = \bigvee_{v \in V_{nc}(q)} \left(\bigwedge_{\pi \in v} \bigcirc \pi \wedge \bigwedge_{\pi \in \mathcal{X}_{nc} \setminus v} \neg \bigcirc \pi \right),$$

be the Boolean representation of $V_{nc}(q)$ at the next execution step.

Our goal is to find, for each $q \in Q_c$, a subset $Q_{cut} \subseteq Q_c$ for which the result

$$B'_{nc}(q) \wedge B_{env}(q) \wedge B_{robot}(q) \wedge \neg WP_{env}\big|_{\mathcal{X}'_c, \mathcal{Y}'} \neq \texttt{False} \tag{3.17}$$

is obtained, where WP_{env} is the set of winning positions for the environment and where $(\cdot)|_{\mathcal{X}_c, \mathcal{Y}}$ denotes the *existential abstraction* with respect to propositions in $\mathcal{X}'_c$ and $\mathcal{Y}'$. Consequently, for any $q \in Q_{cut}$ there is valid environment input assignment at the time step after visiting state q that is *not winning* for the environment. One can think of Q_{cut} as being those counterstrategy states where it is possible that the environment has been able to "cut away" a command that will allow the system to proceed to its next goal by applying some environment

input. Moreover, for all such environment inputs that are losing for the environment, there exists a valid command the system may take that is winning for the system.

Using Q_{cut}, we formulate a set of liveness assumptions that restrict the environment from *always* behaving in a manner that prevents the system's progress toward its goals. Notice that Q_{cut} contains all states for which there is an environment and system move not in the environment's strategy; however, not all such moves are necessarily winning for the system player. For instance, a state in Q_{cut} could yield an environment input that does not allow the environment player to move strictly closer to its goal yet only allow system moves that place the system further away from its goal.

We therefore form a set $P_{cut} \subseteq Q_{cut}$ for which the system has safe commands that are winning for the system. We use Q_{commit} (from the deadlock counterstrategy) to define the set of states where there exist system moves that lead the system closer to its goals. We populate P_{cut} as follows:

$$P_{cut} = \{q \in Q_{cut} \mid \exists v_a \in V_a, \exists q' \in Q_{commit}, \exists q_a \in \delta_a(\gamma^c_X(q), v_a) : \\ \forall \pi \in X_c, \pi \in \gamma^a_X(q_a) \text{ iff } \pi \in \gamma^c_X(q')\}. \tag{3.18}$$

We then apply the environment liveness assumption

$$\square\diamond \bigvee_{q^i \in P_{cut}} \left(B_{robot}(q^i) \wedge \bigwedge_{q^j \in \delta_c(q^i, \gamma^c_X(q^i))} \bigcirc \neg B_{env}(q^j) \right). \tag{3.19}$$

This liveness formula disallows the environment from denying the system from taking action that lead it closer to its goals, when the system is in a configuration where there is such an action to be taken. Because we are targeting a set of states P_{cut} rather than an entire counterstrategy, the conditions (3.19) are less

restrictive than those in [5]. In that case, let SCC_c be the set of states belonging to a *strongly-connected component* (SCC) in A_c determined using Tarjan's depth-first search algorithm [81]. Then, the revisions are as follows:

$$\square\diamond \bigvee_{q^i \in SCC_c} \left(B_{robot}(q^i) \wedge \bigwedge_{q^j \in \delta_c(q^i, \gamma_X^c(q^i))} \bigcirc \neg B_{env}(q^j) \right). \tag{3.20}$$

Nonetheless, that approach is adopted as a fallback if the addition of (3.19) fails to render the specification realizable or if Q_{commit} is empty.

Algorithm 3.3 uses the counterstrategy from Algorithm 3.2 to generate liveness assumptions ψ_g^e restricting transitions to cycles of states preventing the system from fulfilling its goals (livelock). Once a candidate liveness assumption is computed, it is checked in Lines 11–17 to ensure that the system's strategy does not contain a sequence of moves that cause the new liveness condition to be falsified. In such cases, realizable returns `False`, and the candidate liveness is removed. The user may elect to accept or discard this formula; if accepted, it is added to the set of runtime certificates.

Example 3.4. *With the specification φ^{abs} in Example 3.2 along with the deadlock revision* (3.9)–(3.16)*, a new counterstrategy is extracted as pictured in Figure 3.7 that is free of deadlock. The set of winning states for the environment, shaded orange in Figure 3.6(a), includes assignments for the environment variable sen, also visualized in the figure. From this, we may observe that there are four states, q_3, q_5, q_6, q_8 in Figure 3.7 for which there is an alternative assignment to sen, namely sen* =`False`*, that does not satisfy* (3.17)*. In each of the other states in the counterstrategy, sen* =`False` *could be replaced with the alternative assignment sen* =`True`*, yet the result will remain in the environment's winning set. Hence, $Q_{cut} = \{q_3, q_5, q_6, q_8\}$.*

Of these states, q_3, q_6, q_8 are those in which the robot's abstraction has an action (move W) driving the system into configurations matching states within the set Q_{commit}

Algorithm 3.3: Synthesizing livelock revisions for an realizable specification φ^{mod}.

procedure SYNTHLIVELOCKREVISIONS(φ^{mod},$\mathcal{A}_c$,WP_{env},$\mathcal{R}$,Q_{commit})

  if $Q_{commit} \neq \emptyset$ **then** ▷ *Eliminate livelocks*

    $Q_{cut} \leftarrow \{q \in Q_c \mid B'_{nc}(q) \wedge B_{env}(q) \wedge B_{robot}(q) \wedge \neg WP_{env}\big|_{\chi'_c,\gamma'} \neq \texttt{False}\}$

    $P_{cut} \leftarrow$ Eq. (3.18)

    $\psi^e_{g,cand}, \psi^e_g \leftarrow$ Eq. (3.19)

  else

    $\psi^e_{g,cand}, \psi^e_g \leftarrow$ Eq. (3.20)

  end if

  if $\neg(\varphi^e \wedge \varphi^{e,a}_{t,g} \wedge \psi^e_g \wedge \psi^e_t \wedge \psi^e_{g,cand})$ **then**

    $\psi^e_g \leftarrow \psi^e_g \backslash \psi^e_{g,cand}$

  else

    $\varphi^{try} \leftarrow$ Eq. (3.2)

    $realiz \leftarrow \mathsf{realizable}(\varphi^{try})$

    if $\neg realiz$ **then** ▷ *System falsifies environment liveness*

      $\psi^e_g \leftarrow \psi^e_g \backslash \psi^e_{g,cand}$

    end if

  end if

  for all $R_i \in \mathcal{R}$ **do** ▷ *User feedback*

    print Liveness revisions found for region R_i.

  end for

  if user accepts livelock revisions **then** ▷ *Add to the specification and attempt to synthesize a new counterstrategy*

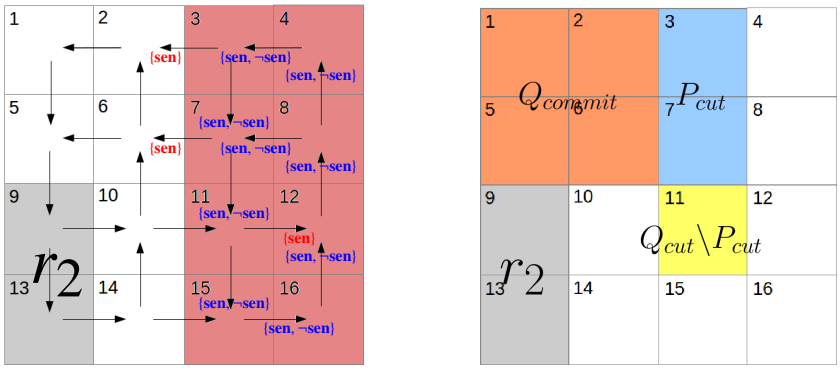
    $\varphi^{mod\prime} \leftarrow$ Eq. (3.2)

    $(realiz', \mathcal{A}'_c, WP'_{env}) \leftarrow \mathsf{ctrStrategy}(\varphi^{mod\prime})$

  end if

  return $realiz'$,$\mathcal{A}'_c$,WP'_{env},$\varphi^{mod\prime}$

end procedure



(a) (b)

Figure 3.6: (a) Partial visualization of the set of winning positions WP_{env}, showing all assignments to the environment proposition *sen* that are in the set WP_{env} in the next step in the execution given the system is currently occupying positions in the red-shaded cells and activating the indicated actions. (b) Map showing regions associated with cut states from Example 3.4.

computed in Example 3.3. The intersection of Q_{cut} *and* $\{q_3, q_6, q_8\}$ *are collected in* P_{cut}*, corresponding to regions shaded blue in Figure 3.6(b). Note that this leaves out* $\{q_5\}$ *(shaded yellow) for which there is an environment move keeping it from immediately realizing an environment goal* ($\square\diamond(\neg sen)$) *but does not lead the system closer to its goal of reaching* r_2*.*

With P_{cut}*, we apply environment liveness revisions* ψ_g^e*:*

$$\square\diamond((x7 \wedge W \wedge \bigcirc\neg sen) \vee (x3 \wedge W \wedge \bigcirc\neg sen) \vee (x7 \wedge S \wedge \bigcirc\neg sen))\,. \quad (3.21)$$

Adding this final revision produces a specification φ^{mod} *that is realizable.*

Proposition 3.2. *The alternating application of the approaches in Algorithms 3.2 and 3.3 in the context of the flow diagram of Figure 4.5 terminates with either a realizable specification or failure.*

Proof. Unrealizability of φ_{abs} with φ realizable implies an inconsistency between the task and the abstraction S_a. The special structure of the problem gives rise to

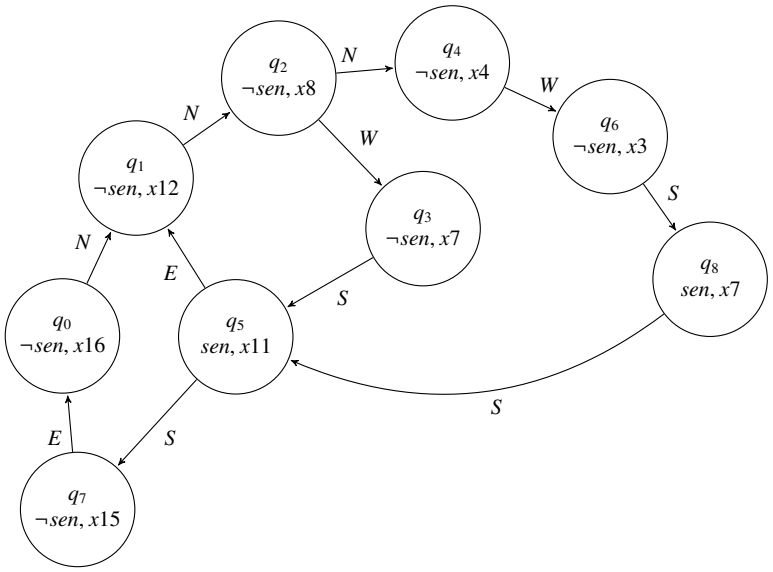


Figure 3.7: Deadlock-free counterstrategy for Example 3.4.

specific selection of the propositions $\mathcal{X}_c \cup \mathcal{Y}$ for B_{robot} and $\mathcal{X}_{nc}$ for B_{env}. Addition of a safety formula (3.6) or a liveness formula (3.19) restricts the behavior of $\mathcal{X}_{nc}$ without altering the transitions of S_a. Since the user may preempt the revision process at any step in the algorithm, we assume without loss of generality that the revisions are always accepted.

We prove soundness by showing that, whenever the revisions ψ_g^e and ψ_t^e are found at any iteration $m > 0$, then either $\mathcal{A}_c^m \neq \mathcal{A}_c^{m-1}$ or the revisions falsify the antecedant of $\varphi^{mod,m-1}$, the modified specification φ^{mod} at step $m-1$. The counterstrategy $\mathcal{A}_c^{m-1}$ produces executions σ_c that are either finite (end in deadlock) or infinite (enter an SCC). If $\mathcal{A}_c^{m-1}$ contains deadlock states, then (3.4) will encode a transition for one such σ_c. If ψ_t^e of (3.5) does not falsify the antecedant of $\varphi^{mod,m-1}$, then this σ_c will not be an execution accepted by $\mathcal{A}_c^m$. If $\mathcal{A}_c^{m-1}$ contains no deadlock states, it must contain an SCC [5] with an execution σ_c accepted by $\mathcal{A}_c^{m-1}$. Thus, either formula (3.19) or (3.20) will produce a ψ_g^e that, if it does not

falsify $\varphi^{mod,m-1}$, then at least one execution trace σ_c will not be accepted by $\mathcal{A}_c^m$.

Further iterations of the main loop in Algorithms 3.2 and 3.3 that uncover new revisions only adds to the executions σ_c removed and, by application of the induction hypothesis, this results in termination either by recovering a realizable specification, or $\mathcal{A}_c^m = \mathcal{A}_c^{m-1}$ (failure). □

Note that the existence of a deadlock or livelock revision is predicated on the reachability of the system goals φ_g^s.

3.5 Creation of Runtime Certificates

Our feedback to the user represents a certificate that, if upheld, will guarantee the mission under the dynamics. We build this certificate around three types of statements: a *command* given to express the added environment safety revisions, a *consequence* used to describe the outcome of the system safety revisions the system's behaviors, and a *cause* for the revisions stemming from the discrete abstraction. We combine the statements automatically into a template of the form

Because [Cause], then [Command]. If these conditions are upheld, [Consequence].

Given this information, the user may choose to accept either the command or consequence statements depending on their consistency with the original design intent. We describe the process for translating the abstraction and revisions into such statements. We also provide a graphical tool that aids this process by allowing the user to interact with a map of the workspace.

3.5.1 Parsing the Revisions

By exploiting the iterative aggregation procedure in Section 3.4.1, we group statements in terms of commit states resulting in certificates that are simpler to parse than would be the case if separate statements were to be given for each region of the configuration space. For each labeled workspace region $R_i \in \mathcal{R}$, we mark those deadlock states (if any exist) from whose predecessors there exists a command $v_a \in V_a$ (Definition 3.1) to reach R_i. Those marked as deadlock states are collected in the set $P_{dead}(R_i)$, defined formally as:

$$P_{dead}(R_i) = \{q \in Q_{dead} \mid \forall q' \in \delta^{c^{-1}}(q), \exists v_a \in V_a, \exists q'_a \in \delta_a(\gamma^c_{\mathcal{X}}(q'), v_a) : \\ \forall \pi \in \mathcal{X}_c, \pi \in \gamma^a_{\mathcal{X}}(q'_a) \text{ iff } \pi \in \gamma_a(R_i)\}.$$

In addition to giving the user a graphical representation for the set of the configuration space over which the revisions have been generated (discussed in Section 3.5.2), we verbally provide the user with a conservative metric for this set. In the fixed point computation in Algorithm 3.1, we keep track of each deadlock state reachable from each state added to Q_{commit}. We use this stored information to find the distances associated with the regions for each state stored in Q_{commit}, and provide the user with a simple numerical metric overapproximating the conditions under which the environment would be required to adhere for the generated revisions to be satisfied. Specifically, this overapproximation is a radius of an enclosing circle projected onto the Cartesian subspace.

Given counterstrategy state $q \in Q_c$, let $[\![\gamma^c_{\mathcal{X}}(q)]\!]$ denote the projection of the region R_i for which $\pi_i = \gamma^c_{\mathcal{X}}(q)$ onto $\mathbb{R}^3 \cap \mathcal{W}$. For each R_i corresponding to a configuration for the system that satisfies some deadlock state in Q_{dead}, we find the relative proximity to a deadlock condition (in terms of physical coordinates)

by finding the maximal pairwise distance between any states affected by the deadlock revisions:

$$(q_i^\star, q_{dead,i}^\star) = \underset{\substack{q \in Q_{commit}, \\ q' \in Q_{reach}(q) \cap P_{dead}(R_i)}}{\arg\max} \left| [\![\gamma_X^c(q)]\!] - [\![\gamma_X^c(\delta^{c^{-1}}(q'))]\!] \right|. \tag{3.22}$$

Here, $|v_{x'}|$ is the Euclidean norm of the real-valued abstraction state $q_a \in Q_a$ represented by a set of propositions $v_{x'} \subseteq \mathcal{X}_c$ that are `True` in that state. The pair of counterstrategy states $q_i^\star$ and $q_{dead,i}^\star$ are those corresponding to a revision for region R_i where the distance is greatest, under the constraint that $q_{dead,i}^\star$ is a deadlock state that is reachable from $q_i^\star \in Q_{commit}$. Note that the distance between the configurations of the two states is:

$$dist_i = \left| [\![\gamma_X^c(q_i^\star)]\!] - [\![\gamma_X^c(\delta^{c^{-1}}(q_{dead,i}^\star))]\!] \right|.$$

Translating Environment Assumptions into Command Statements

Our goal is to provide users with statements such as "Keep sensor *sen* `False` if the robot enters to within N meters of $r2$". We correlate each unique region R_i to the environment proposition assignments prevented by the safety assumption revisions ψ_t^e. Those prevented assignments are given in the formula $\psi_2(q_{dead,i}^\star)$. That is, the added environment assumptions prevent the environment from triggering the combination $\psi_2(q_{dead,i}^\star)$. The data provided to the user is represented by the triple $(R_i, dist_i, \psi_2(q_{dead,i}^\star))$. The triple can be displayed to the user as follows: "If the robot is within $dist_i$ of region R_i, then the generated deadlock revisions (for a given counterstrategy) will be satisfied if the environment is not set to $\psi_2(q_{dead,i}^\star)$." Note that this metric supplies a sufficient but not necessary condition for satisfying the revisions. That is, there might be executions where the system enters within $dist_i$ with any environment setting yet still be able to satisfy the revision formulas.

Translating System Guarantees into Consequence Statements

In a similar manner to our formation of command statements, we generate consequential statements based on the set of states that are backward-reachable from deadlock states. Such statements are used to convey the added guarantees that are required to recover behavior that is close to that of the topology graph. The triple $(R_i, dist_i, \psi_2(q^\star_{dead,i}))$ yields the statement "If the environment does not set $\psi_2(q^\star_{dead,i})$, then the robot will not move to within $dist_i$ of region R_i." The user can elect to accept or discard these statements (hence controlling whether or not ψ^s_t is inserted into (3.2)), providing a more refined level of control over the behavior of the system in the presence of the added environment assumptions.

Translating the Abstraction into Causal Statements

Statements are also provided to the user in order to convey to the user the root cause of the added environment assumptions. These are generated as a consequence of the reachability analysis described in Section 3.4.1. Specifically, the pair $(R_i, dist_i)$ produces the statement "If the robot is within $dist_i$ of region R_i, then it cannot avoid ultimately entering R_i.".

The following example illustrates the procedure.

Example 3.5. *In the result of Example 3.3, let q_{dead} be the deadlock state computed by the counterstrategy corresponding to the configuration $x9$, and designate $Q_{commit} = \{q_1, q_2, q_3, q_4\}$ as the set of commit states for this deadlock. For workspace region r_2, $P_{dead}(r_2) = q_{dead}$, and $Q_{reach}(q_i) = q_{dead}$ for $i = 1, \ldots, 4$. We next determine the pair*

$(q_2^\star, q_{dead,2}^\star)$ *to be*

$$(q_2^\star, q_{dead,2}^\star) = \arg\max \left\{ \left\| \begin{pmatrix} 0 \\ 1 \end{pmatrix} - \begin{pmatrix} 0 \\ 2 \end{pmatrix} \right\|, \left\| \begin{pmatrix} 0 \\ 0 \end{pmatrix} - \begin{pmatrix} 0 \\ 2 \end{pmatrix} \right\|, \left\| \begin{pmatrix} 1 \\ 0 \end{pmatrix} - \begin{pmatrix} 0 \\ 2 \end{pmatrix} \right\|, \left\| \begin{pmatrix} 1 \\ 1 \end{pmatrix} - \begin{pmatrix} 0 \\ 2 \end{pmatrix} \right\| \right\} = (q_2, q_{dead}),$$

where the subscript 2 in the $\star$ *variables is used to signify the fact that the variables apply to region* r_2*. Assuming* $\eta = 1m$*, the corresponding distance is* $dist_2 = \sqrt{1^2 + 2^2} = 2.2m$*. Finally, reflective of the revisions in* (3.9)–(3.16)*, we note the subformula* $\psi_2(q_{dead,2}^\star) =$ *sen.*

Therefore, the generated causal statement is: "the robot cannot avoid entering r_2 *if it enters to within* $2.2m$ *of it". LTL formulas in* (3.9)–(3.12) *are summarized as: "if the robot enters to within* $2.2m$ *of* r_2*, the environment must not set the variable sen to* `True`*". Likewise, the LTL formulas in* (3.13)–(3.16) *are summarized as: "if the environment sets sen to* `True`*, the robot will not enter to within* $2.2m$ *of* r_2*".*

3.5.2 Graphical Visualization Tool

We aid the user in eliciting the runtime certificates generated above via a graphical user interface (GUI), pictured in Figure 3.8, which runs as an extension to the LTLMoP toolkit[2]. The tool allows a user to make queries on different regions and actions with the aid of a map to discover any runtime certificates that have been generated. The generated statements will change depending upon the user's decision to accept or reject the revisions. In the example shown, the region denoted E_dropoff_L indicates an overapproximation and projection of the deadlock-committed states, which has been computed by analysis (e.g. [25]).

In the query example shown in Figure 3.8, the user has selected the region

[2]https://github.com/VerifiableRobotics/LTLMoP/

E_dropoff_L to query the case where the robot is currently in E_dropoff_L. The user has next selected dropoff_L to indicate that moving to dropoff_L is the commanded action the robot is taking. The certificates for this query is provided in the GUI text box. The GUI aids a user when there are numerous certificates to consider, or when the workspace has been re-partitioned as a consequence of reachability analysis. We examine such a use case in further detail in Section 3.6.

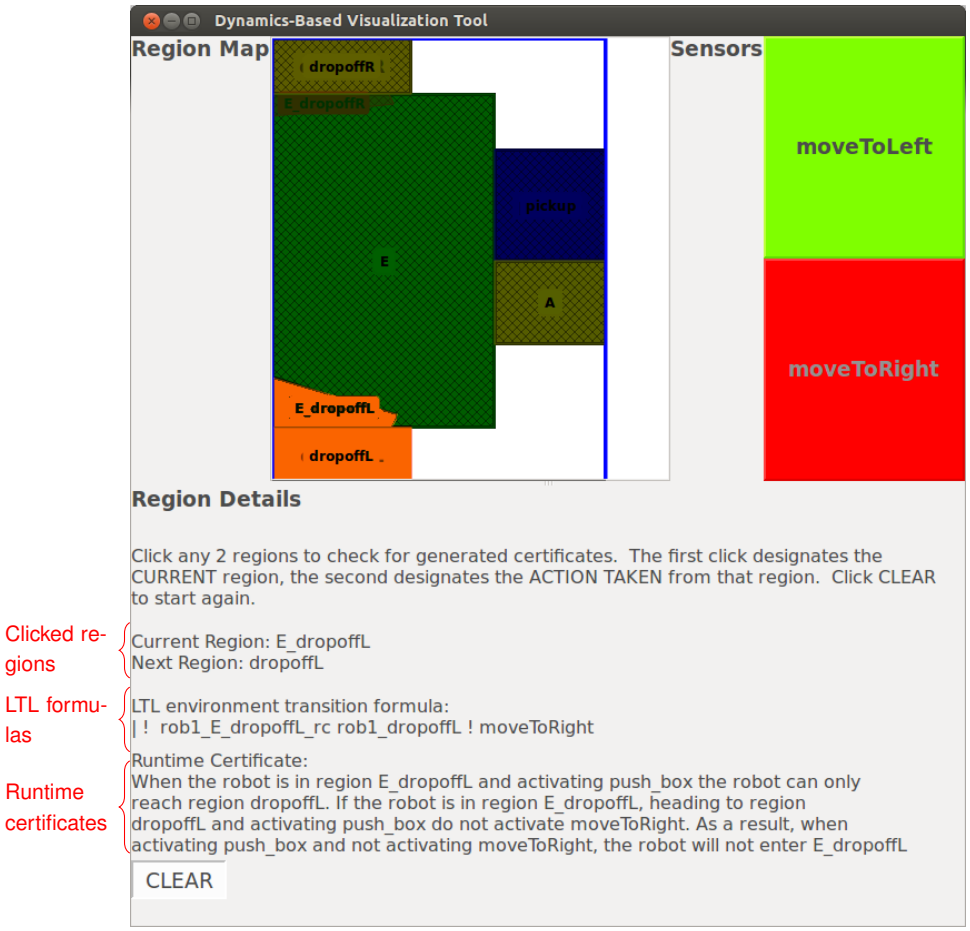


Figure 3.8: A screen capture of the certificate visualization tool. The user specifies the current region and action by clicking regions in the workspace map (highlighted orange). Based on this input, the LTL formulas representing the revision matching that selection are displayed in the info box, along with a certificate of the revisions, provided as text-based feedback. Any sensors that are disallowed as per the command statement are painted red in the upper-right list.

3.6 Case Studies

In this section, we demonstrate the revisions approach in two example scenarios. The examples serve to demonstrate the effectiveness of our approach when the designer is faced with different specifications and different types of discrete abstractions representing the physics of a mobile robot.

3.6.1 Abstractions

We begin by defining two special cases of abstraction that fit in the general definition of Definition 3.1.

Temporally-Grounded Abstractions

The discrete abstraction in a temporal paradigm. By adopting the approach in [71, 76, 95], we may discretize the bounded configuration space $\mathcal{W} \subset \mathbb{R}^n$ and the bounded space of command inputs $\mathcal{U} \subset \mathbb{R}^m$, then define $q'_a \in \delta_a(q_a, v_a)$ to be the set of configurations $\gamma^a_X(q'_a)$ that are reached after some elapsed time τ. Specifically, we denote $[\mathcal{W}]_\eta$ and $[\mathcal{U}]_\mu$ to be, respectively, the uniform grid on $\mathcal{W}$ discretized with resolution η and $\mathcal{U}$ discretized with resolution μ. This grid is defined as follows:

$$[\mathcal{W}]_\eta := \{x \in \mathcal{W} \mid \exists k \in \mathbb{Z}^n : x = k\eta\}, \tag{3.23}$$

$$[\mathcal{U}]_\mu := \{u \in \mathcal{U} \mid \exists k \in \mathbb{Z}^m : u = k\mu\}. \tag{3.24}$$

Note that, in Definition 3.1, $Q_a = [\mathcal{W}]_\eta$, $V_a = [\mathcal{U}]_\mu$. Also, rather than δ_a being defined as the region under which motion completes, it is defined to complete

after some time interval τ. The reader is referred to [19] for full details on the approach to constructing such an abstraction.

Abstractions Grounded on Activation/Completion of Motion

In this case, the transition relation δ_a is defined to be the possible regions $\gamma^a_X(q'_a)$ that may be reached under action $v_a \in V_a$ from region $\gamma^a_X(q_a)$. For a particular transition under δ_a to be defined, the underlying system must guarantee that, under action v_a, the system eventually reaches the set $\gamma^a_X(q'_a)$, $q'_a \in \delta_a(q_a, v_a)$, from any continuous state $\xi \in \gamma^a_X(q_a)$ and remains within the union of $\gamma^a_X(q'_a)$ and $\gamma^a_X(q_a)$. Alternatively, a subset of $\gamma^a_X(q_a)$ may be taken as long as a controller composition property holds (see [25] and references therein for details).

3.6.2 Revisions in a Finely-Partitioned, Temporal Abstraction

In this case study, we return to the factory scenario in Example 3.1 using the workspace in Figure 3.1. To carry out this task, we select a robot described by a unicycle model that is governed by the kinematic relationship:

$$\dot{x} = v\cos\theta, \qquad \dot{y} = v\sin\theta, \qquad \dot{\theta} = \omega,$$

where the x and y are the Cartesian displacements in meters, θ is the orientation angle, and v and ω are, respectively, the forward and angular velocity inputs to the system. The car model is subjected to the constraint where it may only move with *positive* forward velocity (it cannot stop). An abstraction is generated for the three-dimensional configuration space and two-dimensional input space consisting of 2.2×10^6 states, with the chosen values $\eta = 0.15$, $\mu = 0.2$, $\tau = 0.35$.

We generate a temporally-grounded abstraction, as described in Section 3.6.1, using the Pessoa Toolbox.[3] For synthesis, we use the Slugs Synthesis Tool, part of the LTLMoP Toolkit;[4].

The general specification is realizable, producing the controller pictured in Figure 3.9; however the specification φ^{abs} (with respect to the unicycle model) is unrealizable. With the approach in Algorithms 3.2 and 3.3, we compose revisions that render φ^{mod} realizable. After a counterstrategy is synthesized, revisions are found for a total of 2040 states in the counterstrategy (taking 1020 seconds to synthesize on a laptop PC with a dual-core processor and 8GB memory). A metric for these revisions is generated and the user is prompted with the following:

```
Deadlock revisions found.
When within 1.32 m of station_1, never set environment variable
s1_occupied to True.
Accept?  (y/n)
If the environment variable s1_occupied is set to True, then the
robot will never enter to within 1.32 m of station_1.
Accept?  (y/n)
```

Note that, as our configuration space consists of variables of mixed units, the norm computed in (3.22) has been projected onto the Cartesian plane. Recall that accepting the first statement (environment behaviors) removes deadlocks associated with entry into station_1, while accepting the revisions to the second statement does not alter the realizability of the resulting specification, but instead produces different system behaviors in proximity to station_1.

[3] https://sites.google.com/a/cyphylab.ee.ucla.edu/pessoa/
[4] https://github.com/VerifiableRobotics/slugs

A second prompt is given:

```
When within 1.44 meters of station_2, never set environment
variable s2_occupied to True.
Accept?  (y/n)
If the environment variable s2_occupied is set to True, then the
robot will never enter to within 1.44 m of station_2.
Accept?  (y/n)
```

At this point, should the user accept both revisions, a new counterstrategy is synthesized containing no deadlock states. The user is prompted again:

```
Livelock revisions found.  When within 1.35 meters of station_1,
always eventually set environment variable s1_occupied to False.
Accept?  (y/n)
Livelock revisions found.  When within 1.46 meters of station_2,
always eventually set environment variable s2_occupied to False.
Accept?  (y/n)
```

Note that the revisions are associated with a set of counterstrategy states in P_{cut}. The locative commands are computed in a similar manner to the deadlock revisions by replacing the states within P_{cut} in place of Q_{commit}.

This time, the specification is realizable if the user accepts this revision. The resulting execution for the controller is as shown in Figure 3.10. The trajectories pictured in the figure represent evolutions of the continuous nonlinear system when commanded by the synthesized controller. Forward integration is applied to solve the equations of motion using an integration step size of 0.001 sec. Note that the system in the figure infinitely often visits the three regions and is able

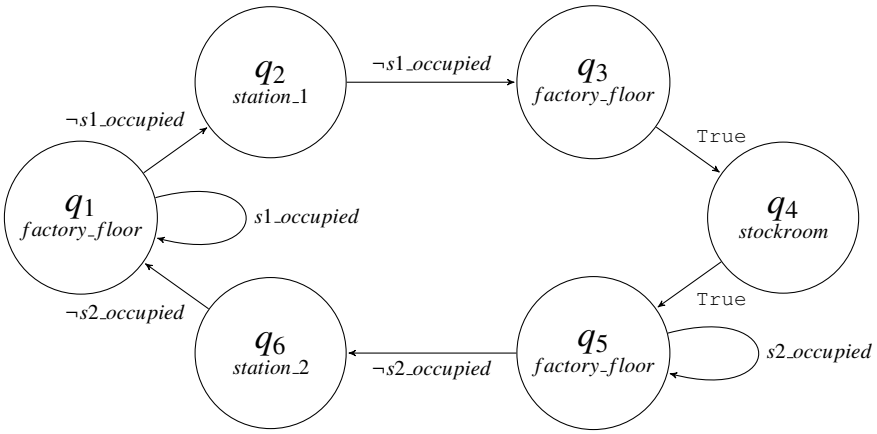
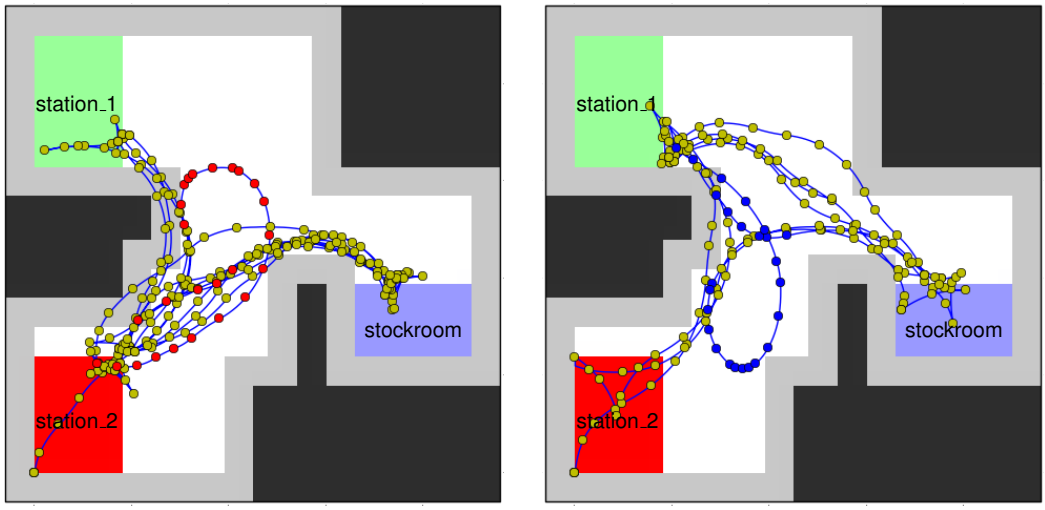


Figure 3.9: Controller for φ in Example 3.1. Edges are labeled with the disjunction of assignments in X that may be assumed for that transition.

to react to a change in the environment. In Figure 3.10(a), the system avoids the region station_1 whenever s1_occupied turns True, this happens at distances greater than 1.32 m of station_1. A similar result is seen in Figure 3.10(b). These behaviors are consistent with the intended behaviors encoded by the specification in Example 3.1.

3.6.3 Workspace Re-Partitioning in the Activation/Completion Paradigm

In this case study, we examine the use of our approach to fulfill a specification where the robot is tasked with moving packages from a pick-up area to one of two drop-off locations, as pictured in Figure 3.11. The specification is, "Visit the loading area. If push_box is active and go_to_left is requested, visit dropoff_L. If push_box is active and go_to_right is requested, visit dropoff_R. Activate push_box when in pickup and deactivate push_box when in dropoff_L or



(a) (b)

Figure 3.10: Continuous trajectories for the nonlinear unicycle abstraction in a 5×5 workspace, where the robot is initialized at the lower-left corner of the workspace. Dots along the trajectory indicate the position of the robot when a new control command is received (a time step of 0.35 seconds). Color indicates the state of the environment (red: s1_occupied; blue: s2_occupied). (a) shows a trajectory when the s1_occupied sensor is activated. (b) shows a trajectory when the s2_occupied sensor is activated.

dropoff_R."

We employ a KUKA youBot to perform the task, which operates on an omni-directional base whose position and orientation is measured in real time. Packages are moved by way of pushing them along the ground using the robot's front fender. There is one action (treated as a system variable) in this scenario, push_box, which is `True` whenever the robot is moving a package and `False` otherwise.

The discrete abstraction for the robot is created using the activation/completion approach of Section 3.6.1. When pushing the box, we impose conditions of the under-actuated unicycle model, constrained with fixed forward velocity,

as explained in Section 3.6.2 in order to assure that the box always maintains contact with the robot. When the robot is no longer required to push the box or when it must disengage with the box, holonomic (fully-actuated) dynamics are imposed to allow the robot to move freely.

We synthesize, using the procedure in [25], a finite-state machine satisfying the mission specification consisting of 6 states and 9 transitions. Controllers were synthesized for the dynamical system using a low-level controller synthesis procedure described in [25] that uses the FSM resulting from synthesis of the general formulas to compute controllers that respect reachability under the dynamics, guaranteeing any sequence of FSM states from any initial robot state selected within the reachable set. If the process fails to compute a controller for some behavior in the FSM, the workspace is re-partitioned and a change to the discrete abstraction is triggered. If the resulting specification is unrealizable, this prompts a call to the revisions approach discussed in this chapter to uncover any certificates associated with an unrealizable specification.

In this case study, the original specification could not be implemented using the imposed dynamics, necessitating an update to the abstraction and the creation of new regions $R_{middle,L}$ and $R_{middle,R}$ based on reachability computations ($R_{middle,L}$ is as indicated in Figure 3.11). Using the proposed revisions approach and calls to the slugs synthesis tool, we automatically generate runtime certificates that restrict the environment's behavior and alter the system's behavior to accommodate the robot's limited capability for movement in these regions. In Figure 3.8, one such certificate is shown for the case where the robot is in $R_{middle,L}$ and activating motion to dropoff_L. The configuration under consideration are highlighted in orange, and the sensor value (request) that is required to be inac-

tive for that configuration is highlighted in red. A similar set of certificates was generated for $R_{middle,R}$.

The execution of the controllers generated as a result of the generated revisions are shown in Figure 3.11. As the robot is heading to dropoff_L but while still outside $R_{middle,L}$, the environment (human operator) is not violating the certificate if the request is changed from go_to_left to go_to_right, and hence the system is guaranteed to react to the environment and execute the task.

3.7 Conclusions

In this chapter, we have described an automatic approach for generating runtime certificates for missions carried out on physical systems with dynamics. Our contribution is an approach that makes use of the problem structure for reactive missions to arrive at certificates that preserve the behavior of the physical system when executing a controller generated from a general specification that is agnostic to the dynamics. The proposed approach features a mechanism for providing feedback to the user as text-based feedback, aided by a GUI, and enables the freedom to accept or reject any such proposed formula at synthesis time. A key benefit of our framework is the ability to generate a small number of revisions for the task, and those that are generated are concise enough to be easily interpreted. This provides the user the best opportunity at synthesizing a controller that is consistent with the original design intent of the specification.

Future work includes providing a means for suggesting a richer set of possible revisions to give as feedback to the user, thereby offering him or her a multiplicity of possible options to apply (e.g. trading off modifying the system's be-

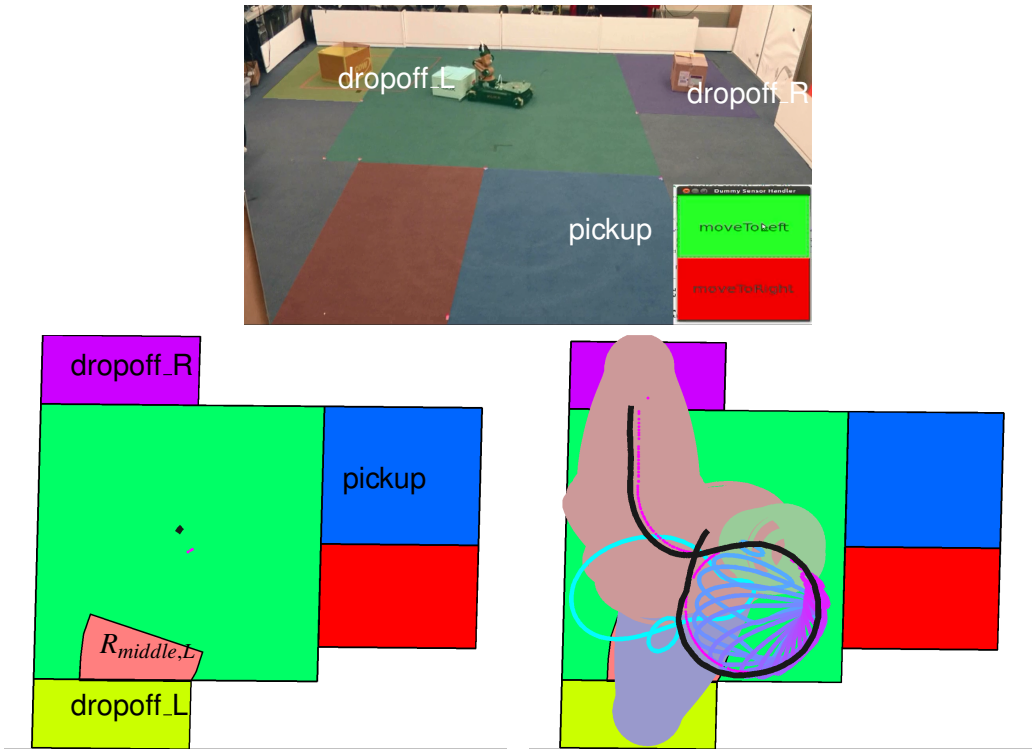


Figure 3.11: Problem set up for the box-transportation scenario. The top image shows the KUKA youBot performing the task of delivering a box to the appropriate region as determined by the sensor. The map is displayed at the bottom left. The new region as a result of the reachability-based re-partitioning, $R_{middle,L}$, is shown in pink. The robot's trajectory is shown in black at the bottom right, along with the reachable sets for the activated controllers, and a nominal trajectory (magenta). A runtime certificate is generated (indicated in Figure 3.8) that indicates that the sensor should not change go_to_left to go_to_right when the robot is in $R_{middle,L}$. For full details, the reader is referred to [25].

havior vs. restricting the environment). Such an extension will involve mining more complex formulas from the synthesis game and automatically translating such formulas into easy-to-understand explanations. Future efforts toward user studies would give the ability to objectively evaluate the effectiveness of the tool as users create their own specifications for (possibly complex) robotic systems.

CHAPTER 4

MULTI-ROBOT REACTIVE MISSION AND MOTION PLANNING AVOIDING DYNAMIC OBSTACLES

4.1 Introduction

Mobile robots, such as package delivery robots, personal assistants, surveillance robots, cleaning robots, mobile manipulators or autonomous cars, execute possibly complex tasks and must share their workspace with other robots and humans. For example, consider the case shown in Figure 4.1 in which two mobile robots are tasked with patrolling and cleaning the rooms of a museum. What makes this task challenging is that the environment in which the robots operate could be filled with static obstacles, as well as dynamic obstacles, such as people or doors, that could lead to collisions or block the robot. To guarantee the task of continuously monitoring all the rooms, each robot must react to the environment at runtime in a way that does not prevent making progress toward fulfilling the overall mission. In particular, we describe an approach for navigation in dynamic environments that is able to satisfy a mission by resolving *deadlocks*, i.e. situations where a robot is temporally blocked by a dynamic obstacle and can not make progress towards achieving its mission, at runtime.

Planning for multi-agent systems has been explored extensively in the past. Many have focused on approaches for local motion planning [1, 87] that offer collision avoidance in cluttered, dynamic environments. While these approaches are effective for point-to-point navigation, the planning is myopic and could fail when applied to complex tasks in complex workspaces. On the other hand, it has been demonstrated that correct-by-construction synthesis from lin-

ear temporal logic (LTL) specifications has utility for composing basic (atomic) actions to guarantee the task in response to sensor events [30,52,58,91]. Such approaches are naturally conducive to mission specifications written in structured English [51], which are translatable into LTL formulas over variables representing the atomic actions and sensor events associated with the task.

In the surveillance-cleaning scenario of Figure 4.1, the motion (moving between rooms), atomic actions (e.g., "remove garbage", "identify a subject"), and binary sensors (e.g. "intruder sensing", "garbage sensing") are assumed to be perfect: they are treated as black boxes that always return the correct result and hence admit a *discrete abstraction* that is appropriate for the task and workspace. A major challenge underpinning this approach is in creating atomic elements holding guarantees for correct execution of the discrete abstraction. To guarantee motion fulfillment, researchers have explored combining LTL-based planners with grid planners [10], sampling-based planners [42], or planners for multiple robots predicated on motion primitives [77]. Such approaches are able to guarantee motion in cluttered environments but do not readily extend these guarantees to cases where the environment is dynamic in nature. Solutions have been sought that, in a computationally expensive manner, partition the workspace finely [60,91] or re-compute the motion plan [10], or else apply conservative constraints forbidding the robot to occupy the same region as an adversarial agent [50].

In the approach introduced in this paper, we alleviate such difficulties by considering an integration of a high-level mission planner with a local planner that guarantees collision-free motion in 3-D workspaces when faced with both static and dynamic obstacles, under the assumption that the dynamic obsta-

cles are not intentionally adversarial. In this context, "intentionally adversarial" means that the dynamic obstacles may behave in a way that may temporarily prevent the robot from achieving a goal, but cannot move in a way that actively always prevents the robot from achieving its goals, for instance by blocking the robot forever. Our integration involves two components: an off-line algorithm for plan synthesis adopting the benefit of an LTL formalism, and an on-line local planning algorithm for executing the plan. Our approach is centralized for the robots in the team, and decentralized with respect to moving obstacles, i.e. we do not control the moving obstacles. While the robots are able to measure the position and velocity of moving obstacles, they only need to do so only within a local range of the robot – the key assumption in this paper is that the robots are not required to have global knowledge of their environment.

The basis of the off-line synthesis is a novel discrete abstraction of the problem that applies simple rules to resolve *physical deadlocks*, between two or more robots in a team or between a robot and a dynamic obstacle. This abstraction is composed with a specification of a multi-agent task to synthesize a strategy automaton encoding the mission plan. In contrast to approaches that would require on-the-fly re-planning upon encountering a physical deadlock [10, 42, 64], the approach we propose automatically generates alternative plans within the synthesized automaton. As with any reactive task, there may exist no mission plan that guarantees the task, due to the conservative requirement that a mission plan must execute under *all possible* environment behaviors. To address this conservatism, our approach automatically identifies for which environment behaviors the mission is guaranteed to hold. These additional assumptions are transformed succinctly into a certificate of task infeasibility that is explained to the user.

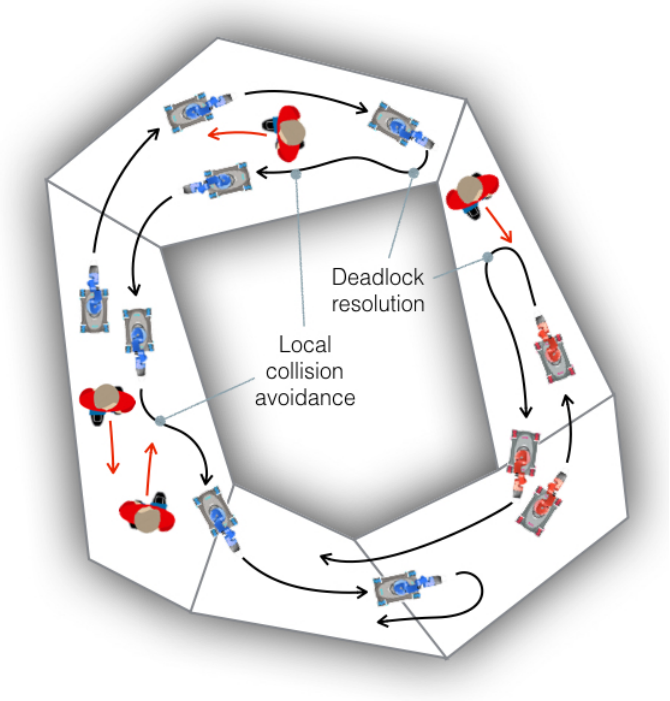


Figure 4.1: Surveillance/cleaning scenario. Two robots are tasked with actively monitoring the rooms of a museum. The robots must avoid collisions with static and moving obstacles and resolve deadlocks in order to achieve their goals.

The on-line execution component is based on a local planner that can optimally avoid dynamic obstacles in two- or three-dimensions, executed as a service called during execution of the strategy automaton. Given a dynamic model of the robots and a coarse description of the moving agents (e.g. their maximum velocities) our local planner computes a plan that guarantees collision-free motion between the robot and static and dynamic obstacles. The collision-avoidance feature obviates the need for collision avoidance to be taken care of by the discrete abstraction. It furthermore allows our local planner to preserve the behaviors of the strategy automaton, by preventing a robot from entering unintended regions as it carries out its task. To the authors' knowledge, this is the first end-to-end system that has been devised to guarantee multi-agent

mission-level tasks in dynamic environments using optimization-based local planners.

The proposed deadlock resolution approach is motivated by works in event-driven planning (e.g. [27]), but yields a strategy that scales well with the number of dynamic obstacles without incurring conservatism that would prevent mission plans from being synthesized. In particular,

- Our approach establishes proof for task success without requiring a costly re-planning step or fine workspace discretization, as long as the environment that causes deadlocks behaves according to the generated assumptions.
- Our approach comes with proof that admissible deadlocks are always resolved and livelocks (the situation where a robot is free to move but unable to reach a goal) never occur.
- The fully automated nature of our approach has practical utility, since the user does not need to intervene to debug specifications. In fact, our approach explains, in an intelligible way, any additional environment assumptions it has added.
- Another practical feature of our approach is that, unlike related planners [50, 60], we do not require global knowledge of the obstacles. As we show, this allows our approach to scale to an arbitrary number of dynamic obstacles, as long as the aggregate behavior of the obstacles adhere to all specified assumptions.

Our approach is suited for dynamic environments that are not actively adversarial; for instance, in human environments where the participants can be

informed of assumptions regarding the robot's assumptions on its behaviors. Specifically, our automatically-generated environment assumptions are transformed into human-readable certificates such as:

The synthesized controller is certified for this task, if any encountered deadlock between the robot and a dynamic obstacle in the hallway resolves eventually.

The certificates provide, at synthesis time, a set of rules defining situations which could make it impossible for the robot to achieve its goals, with the purpose of creating a layer of cooperation between the user (i.e. the human that performs the controller synthesis and deploys the system) and the robots. This frees a user from having to come up with assumptions that characterize the environment's behavior, a difficult proposition in practice. If these assumptions are broken at runtime, then this signifies that the task is no longer strictly guaranteed. Our approach also aims to reduce situations where members of the robot team become deadlocked with one another, by adopting a coordination strategy in the specification preventing actions that may induce deadlocks.

In a preliminary version of this work, [23], a strategy was developed for synthesizing controllers for guaranteed collision-free motion of a robot team. In this paper, we extend those results by presenting a complete description of the proposed abstraction method and off-line controller synthesis procedure, solidify details on the mathematical derivation for the constraints of the local motion planner, and provide in-depth evaluation of our proposed synthesis techniques aided by both simulation and physical experiments. Additionally, we enhance the approach in two ways. First, our approach reasons about the geometry of workspace regions in order to avoid preventable deadlock. For instance, if a corridor is only wide enough for one robot, we offer an approach that coordi-

nates the actions of two robots so that they do not head in opposite directions in the corridor. Second, we present a general approach that allows a richer set of deadlock resolution rules to be chosen at synthesis time.

4.1.1 Related Work

Reactive Synthesis for Mission Planning

A number of approaches are suited to automatic synthesis of correct-by-construction controllers from mission specifications written as temporal logic formulas [10, 42, 61]. Reactive synthesis [52, 91] extends these capabilities to tasks in which the desired outcome depends on uncontrolled events in the environment and changing sensor inputs, and is especially compelling given the complex nature of multi-agent scenarios. For instance, [86] synthesized control and communication for producing optimal multi-robot trajectories, [14] distributed a specification among a robot team, and [72,74] synthesized centralized reactive controllers based on analytically constructed multi-robot motion controllers. Distributed and decomposition-based planning approaches tackle the complexity problem when scaling to a large number of robots. For instance, [83] construct distributed controllers from a specifications already separated into coordinating and non-coordinating tasks, while [78] automatically decompose a specification into independent, distributed task specifications. In contrast, our method only requires local awareness of the robot's surroundings, and guarantees collision-avoidance via a local planner.

Reactive synthesis in dynamically-changing environments presents a crucial dilemma: explicitly modeling the state of all other agents can be computation-

ally prohibitive, but incomplete models of the environment destroy task satisfaction guarantees. To address the state-explosion problem while tracking the state of uncontrollable agents, [92] formulated an incremental synthesis procedure that started with a set number of agents assumed observable, and added more agents to this set depending on available computational resources; however, unlike our approach, they still required global knowledge of the external agents. The authors in [60], on the other hand, made local modifications to the synthesized strategy when new elements of the environment were discovered that violated the original assumptions. While we also update our specification, we differ from [60] in that we synthesize strategies before execution, thereby preserving guarantees at runtime.

Specification Revisions

Recent efforts in reactive synthesis have focused on automatically identifying certain environment assumptions that may prevent the existence of a controller that satisfies the task. Approaches to assumption-mining have provided techniques that enable automatic specification debugging for specifications of any structure [5,55]. While providing the ability to automate the debugging process, they still requires input from the user, for instance the variables the user desires and a final selection of candidate assumptions generated by the algorithm, which has drawbacks for realizing a fully-automated robotic mission planner. An assumption-mining approach to certify the necessary environment assumptions for a given task and robot dynamics was introduced in [21], however, the dynamics-based abstraction do not extend naturally to multi-agent scenarios. This proposed approach obviates the need for the user to intervene during the

planning process.

We propose a novel approach in which assumptions on the environment are generated to identify likely deadlock situations. These added assumptions may be interpreted as restricting the mobility of the uncontrolled agents and are relaxed, when possible, by identifying when they may be violated, if only on a temporary basis. In this regard, our approach is inspired by works on error resilience [33] and recovery [90] in reactive synthesis.

Motion Planning in Dynamic Environments

Collision-free (and deadlock-free) motion planning for multi-robot teams has been successfully demonstrated via non-convex optimization, as proposed in [8, 67], but these approaches did not account for dynamic obstacles, nor could be computed in real-time. On the other hand, convex optimization approaches for collision avoidance, such as [87] and [1], are on-line and account for dynamic obstacles, but cannot reason globally to resolve deadlocks. In this work, we extend these works to enforce collision avoidance and motion constraints over a short time horizon, where the global execution is given by a discrete controller synthesized from a mission specification.

There are two main bodies of work in motion planning that are relevant to our efforts: *collision avoidance* and *deadlock resolution*. The authors of [48] applied pedestrian-avoidance principles to deadlock resolution in narrow passageways. While our approach is similarly reactive to the environment, we additionally reason about situations that cannot be locally resolved (e.g. a blocked corridor). Along similar lines, [15] described a centralized graph search technique for mo-

tion planning, but did not consider dynamic obstacles, and required a rich underlying graph to represent multi-robot motions with kinematic constraints. In contrast, our proposed local planning approach presents a more concise discrete abstraction and also applies to 3D environments.

4.1.2 Contribution

This paper presents three major contributions toward *reactive mission and motion planing with deadlock resolution among dynamic obstacles.*

- A holistic synthesis approach to provably achieve collision-free behaviors in dynamic environments with an arbitrary number of moving obstacles that does not require mutual exclusion. The approach leverages (a) reactive mission planning to globally resolve deadlocks and achieve the specified task, and (b) on-line local motion planning to guarantee collision free motion and respect the robot kinodynamics.
- An automatic means for encoding tasks that resolve deadlock based on automatically-generated revisions to a specification. Our approach automatically generates human-comprehensible assumptions in LTL that, if satisfied by the controlled robots and the dynamic obstacles, would ensure correct behavior. We show that our revision approach is sufficient in making the original specification realizable.
- An optimization-based method for local motion planning that guarantees real-time collision avoidance with *static* and dynamic obstacles in 3D environments while remaining faithful to the robot's dynamics. The method

extends [4] by efficiently computing the robot's local free-space in cluttered environments.

We also present extensive experimental results with ground robots and simulations with aerial vehicles.

4.1.3 Organization

The remainder of this paper is structured as follows. The required concepts for off-line synthesis and on-line motion planning are described in Section 4.2. We formalize the problem in Section 4.3 and give an overview of the method in Section 4.4. In Section 4.5, we introduce a strategy for mission planning for resolving deadlock at runtime, while, in Section 4.6, we introduce an automated approach for generating runtime certificates and a coordination scheme for mission planning. In Section 4.7, we describe the on-line motion planner. We provide theoretical guarantees of the integrated approach in Section 4.8. In Section 4.9, we present extensive simulation and experimental results. Conclusions and future work are provided in Section 4.10.

4.2 Preliminaries

Throughout this chapter, scalars are denoted in italics, x, and vectors in bold, $\mathbf{x} \in \mathbb{R}^n$, with n denoting the dimension of the workspace. The robot's current position is denoted by $\mathbf{p} \in \mathbb{R}^n$ and its current velocity by $\mathbf{v} = \dot{\mathbf{p}}$. A map of the workspace $W \subset \mathbb{R}^n$ is considered, and formed by a set of static obstacles,

given by a list of polytopes, $\mathcal{O} \subset \mathbb{R}^n$. For mission synthesis the map is abstracted by a set of discrete regions $\mathcal{R} = \{R_1, \ldots, R_p\}$, and their topological connections, covering the obstacle-free workspace $F = \mathbb{R}^n \setminus \mathcal{O}$, where the open sets $R_\alpha \subseteq W$.

We consider robots moving in $\mathbb{R}^3$ and approximate them by their smallest enclosing cylinder of radius r and height $2h$, denoted by V. Its ε-additive dilation of radius $\bar{r} = r + \varepsilon$ and height $\bar{h} = h + \varepsilon$ is denoted by V_ε. For a set $X \subset \mathbb{R}^n$ we denote the collision set by $X + V = \{\mathbf{p} \in \mathbb{R}^n \,|\, X \cap V(\mathbf{p}) \neq \emptyset\}$, with $V(\mathbf{p})$ a volume V at position $\mathbf{p}$. Throughout, the notation $\|\cdot\|$ is used to denote the Euclidean norm.

We consider a set of dynamic obstacles DO and denote the volume occupied by a dynamic obstacle $i \in DO$, at position $\mathbf{p}_i$, by $V_i(\mathbf{p}_i)$. To be able to prove safety in dynamic environments, we assume that all moving obstacles either maintain a constant velocity during the planning horizon (a couple of seconds), or that they employ an identical algorithm for collision avoidance as our robots, as introduced in the Reciprocal Velocity Obstacles literature [4]. In this work we do not treat the case where moving obstacles seek collisions and are capable of overtaking the robots. Instead, we assume a fair environment - one where it is always possible for the robots to avoid collisions - such as the case when operating with humans or other risk-adverse agents.

4.2.1 Linear Temporal Logic

LTL formulas are defined over the set AP of atomic (Boolean) propositions by the recursive grammar $\varphi ::= \pi \in AP \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \bigcirc\varphi \mid \varphi_1 \,\mathcal{U}\, \varphi_2$. From the Boolean operators $\wedge$ "conjunction" and $\neg$ "negation", and the temporal opera-

tors $\bigcirc$ "next" and $\mathcal{U}$ "until", the following operators are derived: "disjunction" $\vee$, "implication" $\Rightarrow$, "equivalence" $\Leftrightarrow$, "always" $\Box$, and "eventually" $\Diamond$. We refer the reader to [88] for a description of the semantics of LTL. Let AP represent the set of atomic propositions, consisting of *environment* propositions ($\mathcal{X}$) corresponding to thresholded sensor values, and *system* propositions ($\mathcal{Y}$) corresponding to the robot's actions and location with respect to a partitioning of the workspace. The value of each $\pi \in \mathcal{X} \cup \mathcal{Y}$ is the abstracted binary state of a low-level component. These might correspond to, for instance, thresholded sensor values, discrete actions that a robot can take, or a discrete region (e.g. room in a house).

Definition 4.1 (*Reactive Mission Specification).* *A Reactive Mission Specification is a LTL formula of the form* $\varphi = \varphi_i^e \wedge \varphi_t^e \wedge \varphi_g^e \implies \varphi_i^s \wedge \varphi_t^s \wedge \varphi_g^s$*, with s and e standing for 'system' and 'environment', such that*

- φ_i^e, φ_i^s *are formulas for the initial conditions free of temporal operators.*
- φ_t^e, φ_t^s *are the safety conditions (transitions) to be satisfied always, and are of the form* $\Box\psi$*, where* ψ *is a Boolean formula constructed from subformulas in* $AP \cup \bigcirc AP$*.*
- φ_g^e, φ_g^s *are the liveness conditions (goals) to be satisfied infinitely often, with each taking the form* $\Box\Diamond\psi$*, with* ψ *a Boolean formula constructed from subformulas in* $AP \cup \bigcirc AP$*.*

A strategy automaton that *realizes* a reactive mission specification φ is a deterministic strategy that, given a finite sequence of truth assignments to the variables in $\mathcal{X}$ and $\mathcal{Y}$, and the next truth assignment to variables in $\mathcal{X}$, provides a truth assignment to variables in $\mathcal{Y}$ such that the resulting infinite sequence satisfies φ. If such a strategy can be found, φ is *realizable*. Otherwise, it is *unrealizable*.

Using a fragment of LTL known as *generalized reactivity(1)*, a strategy automata for φ of the form above can be *efficiently* synthesized [11], and converted into hybrid controllers for robotic systems by invoking atomic controllers [52]. These controllers are *reactive*: they respond to sensor events at runtime.

4.2.2 LTL Encoding for Multi-Robot Tasks

We adopt a LTL encoding of a centralized multi-robot mission that is robust to the inherent variability in the duration of inter-region robot motion in continuous environments [75]. Let $AP_{\mathcal{R}} = \{\pi^i_\alpha \mid R_\alpha \in \mathcal{R}\}$ be the set of Boolean propositions representing the workspace regions, such that $\pi^i_\alpha \in AP_{\mathcal{R}}$ is `True` when robot i is physically in R_α for $\alpha \in [1, \ldots, p]$. We call π^i_α in $AP_{\mathcal{R}} \subseteq \mathcal{X}$ a *completion* proposition, signaling when robot i is physically inside R_α. We also define the set $AP^{act}_{\mathcal{R}} \subseteq \mathcal{Y}$ that captures robot commands that *initiate* movement between regions. We call $\pi^i_{act,\alpha}$ in $AP^{act}_{\mathcal{R}}$ an *activation* variable for moving to R_α (but has not necessarily completed motion to R_α). Non-motion actions are handled similarly. Observe that π^i_α and $\pi^i_{act,\alpha'}$ may be true at the same time if robot i is in $R_{\alpha'}$ and is moving toward R_α, where $R_{\alpha'}$ and R_α are adjacent regions. Also note that this is sufficient for the special case π^i_α and $\pi^i_{act,\alpha}$ (the robot stays put). We assume reasonably that non-motion actions are independent of motion, so that actions themselves do not involve moving within any particular region and, if it is possible to execute a particular action within a region, it can be performed anywhere within that region.

We now solidify the semantics of the LTL formulas in the context of robot mission and motion planning. Let T denote a particular fixed time step at which

the strategy automaton is updated with sensory information and supplies a new input to the local planner (as described in Section 4.4.3). A proposition $\pi \in AP$ is `True` at time $t > 0$ iff $\bigcirc \pi \in \bigcirc AP$ is `True` at $t + T$.

Definition 4.2 (***LTL Encoding of Motion*** **[75]).** *A task encoding that admits arbitrary controller execution durations is*

$$\varphi_t^s : \bigwedge_{\substack{\pi_\alpha^i \in AP_\mathcal{R}, \\ i \in [1, n_{robots}]}} \square \left(\bigcirc \pi_\alpha^i \implies \bigvee_{R_\beta \in Adj(R_\alpha)} \bigcirc \pi_{act,\beta}^i \right),$$

$$\varphi_t^e : \bigwedge_{\substack{\pi_\alpha^i \in AP_\mathcal{R}, \\ R_\beta \in Adj(R_\alpha), \\ i \in [1, n_{robots}]}} \square \left(\pi_\alpha^i \wedge \pi_{act,\beta}^i \implies \bigcirc \pi_\alpha^i \vee \bigcirc \pi_\beta^i \right),$$

$$\varphi_g^e : \square \diamond \bigwedge_{\substack{i \in [1, n_{robots}] \\ \pi_{act,\alpha}^i \in AP_\mathcal{R}^{act}}} \left(\left(\pi_{act,\alpha}^i \wedge \bigcirc (\pi_\alpha^i \vee \neg \pi_{act,\alpha}^i) \right) \vee \left(\neg \pi_{act,\alpha}^i \wedge \bigcirc (\neg \pi_\alpha^i \vee \pi_{act,\alpha}^i) \right) \right),$$

where $Adj : \mathcal{R} \to 2^\mathcal{R}$ is an adjacency relation on regions in $\mathcal{R}$ and n_{robots} is the number of robots. The ψ_t^s-formula is a system safety condition describing which actions can occur ($\bigcirc \pi_{act,\beta}^i$) given the observed completion variables ($\bigcirc \pi_\alpha^i$). Formula ψ_t^e captures the allowed transitions ($\bigcirc \pi_\beta^i$) given past completion (π_α^i) and activation ($\pi_{act,\beta}^i$) variables. Formula ψ_g^e enforces that every motion and every action eventually completes (first disjunct) as long as the activation variable is held fixed (second disjunct). Specifically, the second disjunct in this formula allows the system to change its mind for a given action, absolving the environment from having to complete motion for that action. Both ψ_t^e and ψ_g^e are included as conjuncts to the antecedent of φ.

Take, for example, two regions $R1$ and $R2$, arranged as shown in Figure 4.2, with a robot positioned in $R1$ and heading toward $R2$. The system can only take a subset of actions; in this case, it is free to stay in $R1$ or move to $R2$:

$$\square (\bigcirc \pi_{R1} \implies \bigcirc \pi_{act,R1} \vee \bigcirc \pi_{act,R2}).$$

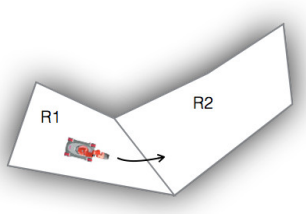


Figure 4.2: Example of two connected regions.

Upon taking an action, for instance move to $R2$ (activate $\pi_{act,R2}$), the system is allowed to be in either of the two regions

$$\square(\pi_{R1} \wedge \pi_{act,R2} \Longrightarrow \bigcirc \pi_{R1} \vee \bigcirc \pi_{R2}),$$

and the environment must eventually allow the system to either arrive at this region or change course

$$\square \diamondsuit ((\pi_{act,R2} \wedge \bigcirc(\pi_{R2} \vee \neg \pi_{act,R2})) \vee (\neg \pi_{act,R2} \wedge \bigcirc(\neg \pi_{R2} \vee \pi_{act,R2}))).$$

To complete the motion encoding, mutual exclusion is also enforced to express the fact that the robot can only be in one region at a time and must decide on one motion at a time. That is, $\square(\pi_{R1} \vee \pi_{R2})$ and $\square(\pi_{act,R1} \vee \pi_{act,R2})$.

We note that it is shown in [31] that complexity of synthesis under the generalized reactivity(1) fragment is polynomial in the size of the state space of the game structure that is, in turn, at most exponential in the total number of propositions. Considering motion alone, the formulas effectively impose restrictions to the allowed state transitions to only consider those that are physically adjacent, effectively reducing the size of the synthesis problem.

4.2.3 Robot Dynamics

Letting $t \in \mathbb{R}_+$ denote time and t_k the current time instant, we define the relative time $\tilde{t} = t - t_k \in [0, \dots, \infty)$ and the time horizon of the local planner $\tau > 0$, greater than the required time to stop if moving at maximum speed. Note that different robots may present different dynamic models.

We denote the state of a robot by $\mathbf{z} = [\mathbf{p}, \dot{\mathbf{p}}, \ddot{\mathbf{p}}, \dots]$, which includes its position and velocity and may include additional terms such as acceleration and orientation. Given a control input $v(t)$ the dynamical model is $\dot{\mathbf{z}} = g(\mathbf{z}, v)$.

Our method is agnostic to the robot dynamics, which are only taken into account by the online local planner. In particular the deadlock resolution and mission planning method is agnostic to the local planner, which can be substituted by a different one that also provides collision avoidance guarantees.

In our local planner, we consider a set of candidate local trajectories, each defined by a straight-line reference $\mathbf{p}_{\text{ref}}(\tilde{t}) = \mathbf{p} + \mathbf{u}\tilde{t}$ of constant velocity $\mathbf{u} \in \mathbb{R}^n$ and starting at the current position $\mathbf{p}$ of the robot. Each motion primitive is then given by an appropriate trajectory tracking controller $\mathbf{p}^{robot}(\tilde{t}) = f(\mathbf{z}, \mathbf{u}, \tilde{t})$ that is continuous in the initial state $\mathbf{z}$ of the robot, respects its dynamical model and converges to the straight-line reference trajectory. Local trajectories are now parametrized by $\mathbf{u}$, see Figure 4.7 for an example. Suitable controllers defining the function $f(\mathbf{z}, \mathbf{u}, \tilde{t})$ include LQR control and second order exponential curves, for ground robots [3] and quadrotors [4].

For fixed robotic platform, controller, initial state $\mathbf{z}$ and reference velocity $\mathbf{u}$, the maximum deviation (initial position independent) between the reference

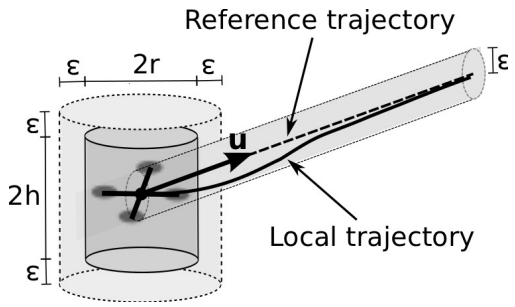


Figure 4.3: Schema local and reference trajectories for an aerial vehicle, generated from the reference velocity $\mathbf{u}$. The tracking error is limited by ε and the robot volume dilated by ε.

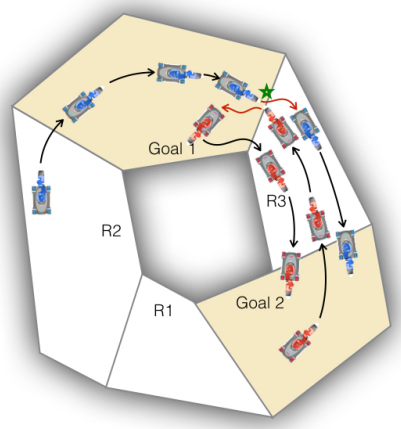
and the simulated trajectory is given by

$$\gamma(\mathbf{z}, \mathbf{u}) = \max_{\tilde{t}>0} \|(\mathbf{p} + \tilde{t}\mathbf{u}) - f(\mathbf{z}, \mathbf{u}, \tilde{t})\|_2. \tag{4.1}$$

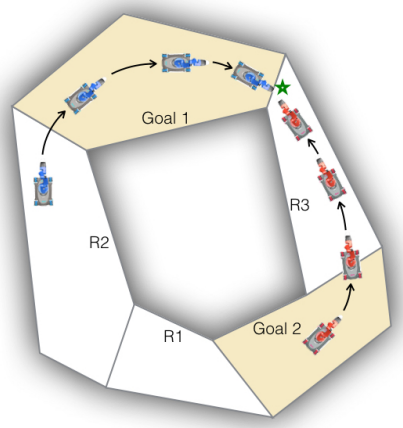
In an off-line procedure, we precompute the maximal tracking errors $\gamma(\mathbf{z}, \mathbf{u})$ via forward simulation of the robot dynamics and controller $f(\mathbf{z}, \mathbf{u}_i, \tilde{t})$ for a discretization of reference velocities $\mathbf{u}$ and initial states $\mathbf{z}$ - we only discretize in initial velocity since the error is independent of the initial position of the robot. They are stored for on-line use in a look-up table.

4.3 Problem Formulation

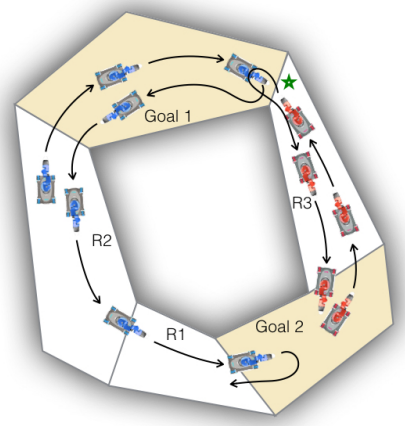
This work combines global planning with local motion planning to produce a correct-by-construction synthesis method that avoids collisions locally yet is able to resolve deadlocks. Synthesis is carried out in a fully-automated way; when modifications to the original specification are necessary, these are explained to the user in an intelligible manner. We provide an example to motivate our correct-by-construction synthesis method.



(a) Specification φ with local planner



(b) Specification φ with local planner



(c) Proposed approach with deadlock resolution

Figure 4.4: Examples of integrated mission and motion planning. The blue robot starts in the region Goal 1 (top) and is tasked to visit Goal 2 (bottom right) and return to Goal 1. The red robot is placed in the region Goal 2 and is tasked to visit Goal 1 and return. The shortest path for both robots, given by solving a specification φ is to go through the corridor on the right. In (a), an execution of a specification φ using a local planner that locally avoids the collision between both robots and succeeds in executing the mission. (b) employs the same specification as (a), but the workspace is shrunk, resulting in a deadlock at location $\star$. (c) shows an execution of a controller synthesized from the modified specification φ'' using the deadlock resolution strategy and local planner developed in this work. With our approach, dynamic obstacles can be avoided locally, as in (a), and deadlocks can also be resolved.

Example 4.1. *Consider the workspace in Figure 4.4(b), where two robots are tasked with visiting regions Goal1 and Goal2 infinitely often; that is,*

$$\varphi_s^g = \bigwedge_{i\in\{1,2\}} \square\diamondsuit(\pi^i_{Goal1}) \wedge \square\diamondsuit(\pi^i_{Goal2}).$$

Figure 4.4 illustrates two approaches for solving this task. Figure 4.4(a) and Figure 4.4(b) show the result of applying a local motion planning scheme to locally avoid collisions with other robots or dynamic obstacles. In certain instances, such as the case shown in Figure 4.4(b), deadlocks can lead to the execution failing to satisfy the task.

Our approach, shown in Figure 4.4(c), relies on a local motion planner to allow several agents per region and avoid dynamic obstacles, as in Figure 4.4(a). Furthermore, it is able to resolve encountered deadlocks that may arise. In this example, when one of the robots encounters deadlock, it reverses its motion to allow the other one to pass into Goal 1, ultimately taking another route to Goal 2.

Definition 4.3 (***Collision).*** *A robot at position $\boldsymbol{p}$ is in collision with a static obstacle if $V(\boldsymbol{p}) \cap O \neq \emptyset$. The robot is in collision with a dynamic obstacle i at position $\boldsymbol{p}_i$ and of volume $V_i(\boldsymbol{p}_i)$ if $V(\boldsymbol{p}) \cap V_i(\boldsymbol{p}_i) \neq \emptyset$.*

Denote by $\mathbf{p}(t)$ the position of a robot at time t and by $\mathbf{p}_i(t)$ the position of a dynamic obstacle i at time t. The trajectory of the dynamic obstacles is estimated between the current time t_k and a time horizon τ. In our model we consider constant velocity.

Definition 4.4 (***Collision Free Local Motion).*** *A trajectory is said to be collision free if for all times between t_k and the time horizon there is no collision between the robot*

and any static or dynamic obstacle,

$$V(\boldsymbol{p}(t)) \cap \left(O \underset{i \in DO}{\cup} V_i(\boldsymbol{p}_i(t)) \right) = \emptyset \qquad \forall t \in [t_k, t_k + \tau]. \tag{4.2}$$

Which is equivalent to

$$V(\boldsymbol{p}(t)) \subset F \qquad \text{and} \qquad V(\boldsymbol{p}(t)) \cap V_i(\boldsymbol{p}_i(t)) = \emptyset \quad \forall t \in [t_k, t_k + \tau], \ \forall i \in DO. \tag{4.3}$$

Definition 4.5 (*Deadlock).* *In this work we consider motion related deadlocks. A robot at position $\boldsymbol{p}$ is said to be in a deadlock if it is not in a collision, it has not achieved the target given by the automaton and it can not make progress towards the goal, i.e. it is not moving, for a prespecified amount of time.*

The goal of this work is to solve a set of problems as follows.

Problem 4.1 (*Local Collision Avoidance).* *Given the dynamics for each robot in the team, construct an on-line local planner that guarantees collision avoidance with static and dynamic (moving) obstacles.*

Problem 4.2 (*Synthesis of Strategy Automaton with Deadlock Resolution).* *Given a topological map, a local motion planner that solves Problem 4.1 and a realizable mission specification φ that ignores collisions, automatically construct a specification φ' that includes both φ and a model of deadlock between robots and unmodeled dynamic obstacles. Use φ' to synthesize a controller that satisfies φ'.*

This synthesized controller will re-route the robots to resolve deadlocks (should they occur), while satisfying the reactive mission specification and remaining livelock free. For mission specifications that consider the presence of possible deadlocks, there may be no satisfying controller. We therefore synthesize environment assumption revisions as additional LTL formulas to identify cases where dynamic obstacles may trigger deadlock and trap the system

from achieving its goals. These formulas are significant because they offer *certificates* explaining the required behaviors of the environment that, if followed, guarantee that the robot team will carry out the task. Such certificates must be conveyed to the user in a clear, understandable manner. An example of such a condition is: "the environment will never cause deadlock if robot 1 is in the kitchen and moving to the door". This leads to the following Problem.

Problem 4.3 (***Revising Environment Assumptions).*** *Given an unrealizable reactive mission specification φ', synthesize environment assumption revisions $[\varphi_t^e]^{rev}$ such that the specification φ'' formed by replacing φ_t^e with $[\varphi_t^e]^{rev}$ is realizable, and provide the user with a human-readable description of these revisions as certificates for guaranteeing the task.*

4.4 Approach

This work solves Problems 4.1, 4.2 and 4.3 via a combined off-line and on-line approach that (a) synthesizes a strategy automaton that realizes the mission and (b) computes a local motion planner that executes the automaton in a collision-free manner. Figure 4.5 highlights the off-line and on-line components and their interconnections, which we now introduce.

4.4.1 Off-Line

The inputs for the off-line part of the method are: (a) a user given mission specification, (b) a discrete topological map of the workspace (which ignores dynamic obstacles) and (c) the dynamic model and controller of the robots in the

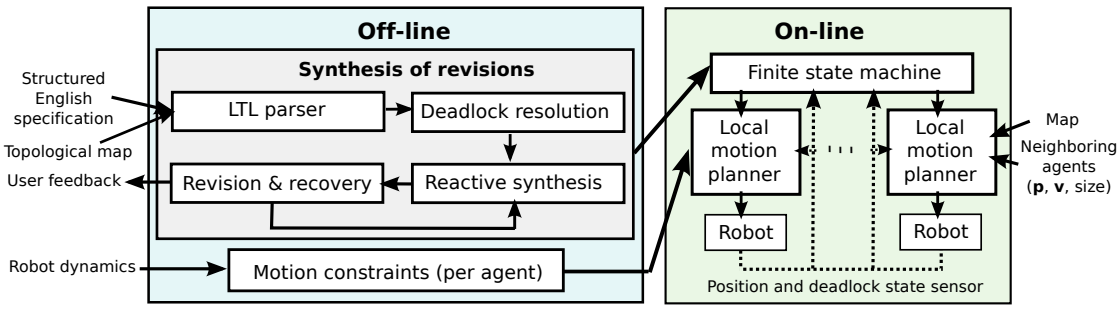


Figure 4.5: Structure of the proposed mission and motion planner, with off-line and on-line parts. The mission planning is off-line and is described in Section 4.5 and in Section 4.6. The motion planner, Section 4.7, is computed at runtime and utilizes the strategy automaton (finite-state machine) synthesized off-line by the mission planner.

team. The off-line part of the method consists of two independent parts.

Mission Planning

In this step we synthesize a centralized controller, or finite state machine, that will guide the robots in the team through the topological map. This controller considers possible *physical deadlocks* between robots in the team as well as with moving obstacles. Since the position of the moving obstacles is not known at synthesis time, environment assumptions are iteratively revised as necessary. The resulting strategy automaton with the revisions included accommodate deadlocks wherever they may occur at runtime, and fulfillment of the specification is guaranteed as long as the environment behaves according to the assumptions explained to the user in the revisions generation step. We also adopt a recovery scheme [90] that synthesizes a strategy that allows violations of environment *safety* assumptions to be tolerated, retaining satisfaction guarantees as long as the violation is transient.

The *mission planning* part of the off-line synthesis approach is described in detail in Section 4.5 and Section 4.6.

Motion Planning

The automaton is agnostic to the robot's dynamics, which are instead accounted for by the local planner. For a given robot model and controller a set of motion constraints, or tracking errors, are precomputed at synthesis time. This part was described in Section 4.2.3.

During execution, the local planner is fed, at runtime, a set of constraints that are then solved for in an efficient manner. These constraints include region boundaries, static and dynamic obstacles and kinodynamic model of the robot.

4.4.2 On-Line

At each time step of the execution, the synthesized strategy automaton provides a desired goal for each controlled robot in the team. Then, each robot independently computes a local trajectory that achieves its goal while avoiding other agents.

If a physical deadlock is sensed, an alternative goal is extracted for the robot from the synthesized strategy automaton. The existence of such an alternative in the automaton is guaranteed by construction if the environment assumptions are satisfied. The local planner builds on [4] by adopting a convex optimization approach as described in Section 4.7.

4.4.3 Integration of mission and motion planning

The proposed method consist of two interconnected parts, the mission planner and the motion planner. Figure 4.5 highlights the components and their interconnections.

The mission planner is computed off-line, prior to execution. It requires a topological map of the environment given by a description of the regions, such as rooms, and their connections. It creates a finite state machine or automaton that achieve the high-level specification and from which the robots in the team can extract a strategy at runtime. Note that we do not optimize the mission planner in this work, but our framework allows us to readily adopt techniques for optimal execution such as [40] to extract an optimal strategy automaton.

At each time instance in the execution, a target motion is extracted from the automaton. The motion planner computes a collision-free motion to make progress towards the target. If a physical deadlock is sensed, an alternative strategy is extracted from the automaton.

The motion planner requires a local map of the environment W, containing all the static and moving obstacles. The regions in the free space F of the local map - used at run-time - must be labeled to match the regions $\mathcal{R}$ of the topological map - used for off-line synthesis.

If the automaton commands a robot to transition between two connected regions, a path is computed from the current position of the robot to the border of the destination region and then is followed by the local planner. If the automaton commands a robot to remain in a region, the local planner moves the robot towards the middle point of the region.

4.5 Off-Line Synthesis: Resolving Deadlock

In this section, we discuss how to synthesize a strategy automaton given a mission specification and a topological map of the environment, provided that, at runtime, a low-level control strategy is applied that guarantees collision-free motion. We assume that the task specification φ ignores collisions, but we allow the possibility that deadlocks can occur at any time during the robot's execution. Deadlocks can trap the robot from achieving its goals, rendering the specification unrealizable. The crux of this work is an approach that systematically modifies the specification with additional behaviors that redirect the robot team in order to resolve deadlocks, whenever possible. If a satisfying mission plan does not exist, the approach iteratively adds assumptions on the deadlock behavior to the specification until a satisfying strategy can be found for the robot team. By focusing on deadlock rather than the positioning of dynamic obstacles, it allows our approach to be valid for any number of dynamic obstacles, as long as they fulfill the stated assumptions returned by our synthesis approach. It also removes the need to globally track the positions of every obstacle at runtime.

An outline of the general approach is shown in Figure 4.6. Such a strategy was chosen to disable any blocked routes to the goal and thereby enable the strategy automaton to seek alternate routes once deadlock has been encountered. In this section, we detail the steps involved to implement the overall approach.

4.5.1 Deadlock Resolution

We declare a robot to be physically in deadlock with another agent if it has not reached its goal but cannot move. This can happen when an agent becomes blocked either by another agent or by a dynamic obstacle. To keep track of which robot is in deadlock, we introduce Boolean input signals $x^{ij} \in \mathcal{X}$, where $i = 1, \ldots, n_{robots}$ and $j = 0, \ldots, n_{robots}$ (the index $j = 0$ representing a dynamic obstacle). Without loss of generality, we consider only deadlock between pairs of agents at a time. For the case where a robot is in deadlock while in proximity to a dynamic obstacle, we let $j = 0$ and refer to this case as *singleton deadlock*. Otherwise, the robot is in deadlock with another robot on its team, $j \neq 0$, and is considered to be in a state of *pairwise deadlock*. The proposition x^{i0} is `True` iff robot i is in singleton deadlock and x^{ij} is `True` iff robots i and j are in pairwise

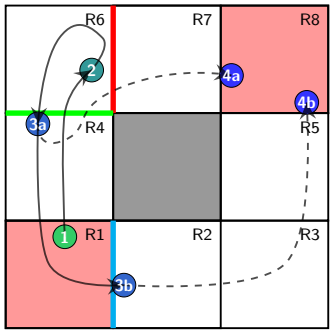


Figure 4.6: Diagram illustrating the deadlock resolution strategy for a single robot tasked with visiting $R1$ and $R8$. Starting in region $R1$ (marked '1'), the robot encounters deadlock (2) in region $R6$, while heading to $R7$. The $R6$-to-$R7$ transition is prevented (red line), and the robot must move a discrete radius m away from the deadlock event to resolve deadlock. If $m = 1$, then deadlock is resolved once the robot crosses the green line, leaving $R6$ (3a). From there, it may reach $R8$ (4a) if no other deadlocks are encountered. On the other hand, when $m = 3$, deadlock is resolved only when crossing the cyan line (3b); an alternate path to the goal may result (4b).

deadlock. We defer detailing our approach for detecting deadlock at runtime to Section 4.7.

To simplify the notation in what follows, we introduce the following shorthand:

$\theta_P^{ij} = \neg x^{ij} \wedge \bigcirc x^{ij}$	rising edge–pairwise deadlock between robots i and j
$\theta_S^{i} = \neg x^{i0} \wedge \bigcirc x^{i0}$	rising edge–singleton deadlock for robot i
$\psi_{\alpha\beta}^{i} = \pi_{\alpha}^{i} \wedge \bigcirc \pi_{\alpha}^{i} \wedge \pi_{act,\beta}^{i}$	incomplete transition ($\alpha \neq \beta$); remain in region ($\alpha = \beta$)

The definition for singleton deadlock is abstract enough to capture the case where deadlock occurs between the robot and any number of dynamic obstacles - singleton deadlock will be set if the robot stops moving when encountering one or more dynamic obstacles blocking its path. On the other hand, since the members of the team are controlled by the same mission planner, pairwise deadlock can be resolved separately. For instance, if three robots on a team converge on the same point, then three pairwise deadlock propositions will be set.

Resolving deadlock by redirecting the robot's motion based on the instantaneous value of x^{ij} alone may result in livelock, where the robot may be trapped from achieving its goals as a result of repeated deadlock status changes. For this reason, our scheme automatically introduces additional memory propositions that are set when deadlock is sensed, and reset once the robot moves a predefined *discrete radius*, denoted m, defining the a deadlock resolution horizon (i.e. it traverses m regions away from the region where deadlock occurred in order for the deadlock to be considered "resolved".

Definition 4.6 (***Discrete Radius).*** *Let $\pi^i_{curr(k_i)} \in AP_{\mathcal{R}}$ and $\pi^i_{act,curr(k_i-1)} \in AP^{act}_{\mathcal{R}}$ be, respectively, the configuration and action taken by robot i, where $k_i = 1, 2, \ldots$ represents*

an event that is incremented when robot i enters a new region, i.e. k_i *is incremented at the time instant when* $curr(k_i - 1) \leftarrow curr(k_i)$*. The current region index* $curr(\cdot) \in [1, p]$ *is defined recursively, initialized such that* $\pi^i_{curr(1)}$ *is the robot's completion when deadlock was recorded and* $\pi^i_{act,curr(0)}$ *is the robot's action when deadlock was recorded. Then, the discrete radius m is the number of successive steps* $k_i \in [1, m]$ *for which we impose the restriction* $\pi^i_{curr(k_i)} \in AP^{act}_{\mathcal{R}} \backslash \{\pi^i_{act,curr(k_i-1)}\}$ *on the robot's actions. This ensures that the robot makes a move that does not re-enter the region just visited.*

The concept behind the proposed deadlock resolution approach is to force the robot to actively alter its strategy to overcome a deadlock by imposing a small number of constraints without directly prescribing the path the robot is required to take. The path is obtained from a strategy automaton synthesized from the specification that has been augmented with any revisions generated by our approach. For instance, as illustrated in Figure 4.6, for the case $m = 1$ (resp. $m = 3$), if a deadlock is sensed at point (2), the revisions forbid the robot from crossing the red line until it reaches the green line (resp. cyan line). As a result, different choices of m will lead to the synthesis of strategies that give rise to different subsequent paths to goal region $R8$ and decisions whether or not to revisit the location where deadlock had occurred.

We first introduce an approach where resolution occurs when the robot leaves its current region, then generalize this approach to allow the user to choose any number of discrete steps, $m \geq 0$, to be taken by the robot before deadlock is declared as resolved. In this work, we assume m to be chosen ahead of time.

4.5.2 Resolving Deadlock when $m = 0$

Our deadlock resolution approach for the case $m = 0$ amounts to the situation where robot i is forced to move in another direction whenever x^{ij} becomes `True` for $j = 0, \ldots, n_{robots}$. As long as x^{ij} remains `True` when robot i is in region R_α, we disallow motion to R_β as follows:

$$\square \bigwedge_{\substack{\pi^i_\alpha \in AP_{\mathcal{R}}, \\ R_\beta \in Adj(R_\alpha)}} \left(\bigcirc x^{ij} \wedge \pi^i_\alpha \implies \bigcirc(\neg \pi^i_{act,\alpha} \wedge \neg \pi^i_{act,\beta}) \right). \tag{4.4}$$

It is easily observed that, as soon as the robot's motion is nonzero when it begins to move in a direction opposite to its previous motion, x^{ij} becomes `False` again and the robot is free to resume its motion to R_β. This can lead to unwanted behaviors, such as chattering. To avoid chattering behaviors, we enrich the deadlock resolution approach to allow for any choice of $m > 0$.

4.5.3 Resolving Deadlock when $m = 1$

For each robot, we introduce into $\mathcal{Y}$ the system propositions $\{y^i_\beta \mid R_\beta \in \mathcal{R}\} \subset \mathcal{Y}$ representing the *deadlock flag* occurring when activating a transition from a given region R_α to region R_β. When the flag is set, the following formula restricts the robot's motion:

$$\square \bigwedge_{\substack{\pi^i_\alpha \in AP_{\mathcal{R}}, \\ R_\beta \in Adj(R_\alpha)}} \left(y^i_\beta \wedge \pi^i_\alpha \implies \bigcirc(\neg \pi^i_{act,\alpha} \wedge \neg \pi^i_{act,\beta}) \right). \tag{4.5}$$

The role of y^i_β is to disallow the current transition (from R_α to R_β), as well as the self-transition from R_α to R_α. The self-transition is disallowed to force the robot to leave the region where the deadlock occurred (R_α), instead of waiting for it to resolve; R_β is disallowed since the robot cannot make that transition.

Next, we encode conditions for detecting *singleton deadlock* at runtime, and storing these as propositions y^i_β that memorize that singleton deadlock had occurred:

$$\square \bigwedge_{\substack{\pi^i_\alpha \in AP_{\mathcal{R}}, \\ R_\beta \in Adj(R_\alpha)}} \left(\neg y^i_\beta \Rightarrow \left((\theta^i_S \wedge \psi^i_{\alpha\beta}) \Rightarrow \bigcirc y^i_\beta\right)\right), \tag{4.6}$$

$$\square \bigwedge_{\substack{\pi^i_\alpha \in AP_{\mathcal{R}}, \\ R_\beta \in Adj(R_\alpha)}} \left(y^i_\beta \Rightarrow \left((\pi^i_\alpha \wedge \bigcirc \pi^i_\alpha) \Leftrightarrow \bigcirc y^i_\beta\right)\right). \tag{4.7}$$

The first formula sets the deadlock flag y^i_β if the robot is activating transition from R_α to R_β. The second formula keepsthe flag set until a transition has been made out of R_α (to a region different from R_β). Notice that, in our construction, singleton deadlock considers deadlock between one robot and *any number of* dynamic obstacles, alleviating the need to globally track or identify obstacles at runtime. While this construction could introduce cycling, we prefer it over an approach that stores the entire path because we can limit the number of propositions added to $\mathcal{Y}$ in order to manage complexity. For instance, if we are aware that deadlock does not occur when the robot is trying to reach a given region $R_\cdot$, we can eliminate the variable $y^i_\cdot$.

For *pairwise deadlock*, we add the following formulas encoding the conditions for declaring that pairwise deadlock has been detected. Note that the disjunction in the formula allows the synthesis tool to decide which one of the two robots should react to the deadlock:

$$\square\left(\theta^{ij}_P \implies \left(\bigvee_{\ell\in\{ij\}} \bigwedge_{\substack{\pi^\ell_\alpha \in AP_{\mathcal{R}}, \\ R_\beta \in Adj(R_\alpha)}} \left(\neg y^\ell_\beta \wedge \psi^\ell_{\alpha\beta}\right) \implies \bigcirc y^\ell_\beta\right)\right). \tag{4.8}$$

We also add the following to ensure that the memory propositions are only set

when the rising edge of deadlock (singleton or pairwise) is sensed.

$$\square\Big(\bigwedge_{\substack{i\in[1,n_{robots}]\\ R_\beta\in\mathcal{R}}}\Big(\neg y^i_\beta \wedge \neg\theta^i_S \wedge \bigwedge_{\substack{j\in[1,n_{robots}]\\ j\neq i}}\neg\theta^{ij}_P\Big) \implies \bigcirc\neg y^i_\beta\Big). \tag{4.9}$$

In practice, we do not need a proposition y^i_β for every $R_\beta \in \mathcal{R}$, but only $d = \max_{R_\alpha\in\mathcal{R}}(|Adj(R_\alpha)|)$ such propositions for each robot in order to remember all of the deadlocks around each region of the workspace. Here $|\cdot|$ denotes the set cardinality. The number of conjuncts required for condition (4.8) is $\binom{n_{robots}}{2}$, but, since the number of formulas contributes at worst linear complexity (due to parsing of each formula), the conjuncts contribute only a small amount to the overall complexity. Note that the complexity of the synthesis algorithm is a function of the number of propositions and not the size of the specification.

Conjuncting the conditions (4.5)–(4.9) with φ^s_t yields a modified formula $[\varphi^s_t]'$ over the set AP, and the new *abstracted* specification $\varphi^{abstr} = \varphi^e_i \wedge \varphi^e_t \wedge \varphi^e_g \implies [\varphi^s_i]' \wedge [\varphi^s_t]' \wedge \varphi^s_g$. The initial conditions are modified by setting the additional propositions x^{ij}, y^i_α to `False`.

4.5.4 Resolving Deadlock when $m > 1$

In some cases, having a deadlock resolution strategy in which multiple discrete steps must be made away from any encountered deadlock may result in different behavior than a strategy in which deadlock is resolved when moving away just one step. Considering Figure 4.6, the case $m = 3$ results in greater exploration of the workspace, whereas the case $m = 1$ results in confinement to a smaller portion of the workspace.

We generalize the strategy presented in Section 4.5.3 by considering the case

where deadlock is resolved once $m > 1$ discrete moves have been taken away from the last encountered deadlock. In what follows, the same formulas as in Section 4.5.3 apply; here, we only describe modifications to this setup. To ensure each robot moves away from deadlock a discrete radius, we require $m - 1$ propositions (for robot i, $y^i_{out,1}, \ldots, y^i_{out,m-1}$) that are set and reset in a chain in order to memorize the robot's position from the encountered deadlock. $y^i_{out,k}$ are initially `False` for all i, k.

In order to set the first such memory proposition in the chain, the terms $\bigcirc y^i_\beta$ in (4.6) and $\bigcirc y^\ell_\beta$ in (4.8) are replaced with $\bigcirc y^i_\beta \wedge \bigcirc y^i_{out,1}$ and $\bigcirc y^\ell_\beta \wedge \bigcirc y^\ell_{out,1}$, respectively, and the abstracted specification φ^{abstr} is constructed based on these formulas. For each subsequent discrete step away from deadlock, we require the remaining propositions to be set when the one with next lowest index has been reset. This behavior occurs through the formula:

$$\square \bigwedge_{k=2,\ldots,m-1} \left(\neg y^i_{out,k} \Rightarrow \left((y^i_{out,k-1} \wedge \bigcirc \neg y^i_{out,k-1}) \Rightarrow \bigcirc y^i_{out,k}\right)\right). \tag{4.10}$$

Additionally, for each $k = 1, \ldots, m-1$, we require that each $y^i_{out,k}$ be reset only when the robot has left the current region; specifically,

$$\square \bigwedge_{\substack{\pi^i_\alpha \in AP_{\mathcal{R}}, \\ k=1,\ldots,m-1}} \left(y^i_{out,k} \Rightarrow \left((\pi^i_\alpha \wedge \bigcirc \pi^i_\alpha) \Leftrightarrow \bigcirc y^i_{out,k}\right)\right). \tag{4.11}$$

Finally, as long as some $y^i_{out,k}$ is set, we also set the deadlock flag memory proposition y^i_α corresponding to the region R_α that the robot had immediately departed. That is,

$$\square \bigwedge_{\pi^i_\alpha \in AP_{\mathcal{R}}} \left((\pi^i_\alpha \wedge \bigvee_{k=1,\ldots,m-1} y^i_{out,k}) \Rightarrow \bigcirc y^i_\alpha\right). \tag{4.12}$$

This prevents the robot from re-entering the region from which it just departed.

The safety revisions restrict the system's moves in the execution sequence be ones that actively take it m away from the location where the deadlock flag was raised. Since waiting in a region is disabled in (4.5), and reentering a region is disabled in (4.12), these safety revisions will cause the system to move m steps away from deadlock in finite time.

In general, setting m large, could lead to behavior that "explores" more of the workspace, but also could result in unrealizability. Consider again the scenario in Figure 4.6, but with R2 always blocked. In this case, $m = 3$ would result in an unrealizable specification because the robot cannot make three discrete steps away from R6 without entering R2. Such design tradeoffs therefore depend on the workspace and its partitioning. Automatic selection of m for a given specification and collection of regions is the subject of future work, as is the use of $\square\diamond$ liveness formulas to resolve livelock in a more direct manner similarly to [5,21] while remaining scalable to the number of robots on the team.

4.6 Off-Line Synthesis: Environment Assumptions and Coordination

If the specification φ^{abstr} is synthesizable, then Problem 2 has been solved and no further modifications to the abstracted specification are necessary. But, the possible presence of humans or other uncontrollable agents in some parts of the environment may cause the abstracted specification to be unrealizable. Then, it becomes necessary to solve Problem 3 to find a minimal set of environment assumptions that restores the guarantees.

We automatically generate assumptions on the environment's behavior in cases where the modified specification is unrealizable. To prevent any unreasonable assumptions (assumptions that the robot can overcome deadlock when it is impossible to do so), we provide a means for coordinating robot actions to prevent such assumptions from being given to the user. Combining the encoding and revisions approach, we formally show that the synthesized automaton is guaranteed to fulfill the task under these assumptions, showing that our approach also removes the possibility of deadlock and livelock from occurring.

4.6.1 Runtime Certificates for the Environment

We note that the dynamic obstacles are uncontrollable agents, and lacking behavioral information, so altering environment assumptions does nothing to characterize their behavior. Rather, we may still provide the user with a certificate under which the environment's behavior will guarantee that the team can achieve all its goals without being trapped permanently in a state of deadlock or livelock. Such assumptions can be given to the user to allow him/her to be mindful of any condemning situations when co-inhabiting the robots' environment. As such, we call these added assumptions *runtime certificates*.

When a specification is unrealizable, there exist environment behaviors (called *environment counterstrategies*) that prevent the system from achieving its goals safely. Here we build upon the work of [5, 21, 55], processing synthesized counterstrategies to mine the necessary assumptions. Rather than synthesize assumptions from the counterstrategy, as in [5], which requires specification revision templates to be specified by hand, we automate the counterstrategy

search by searching for all deadlock occurrences, then store the corresponding conditions as assumptions.

We denote $C_{\varphi^{abstr}}$ as an automaton representing the counterstrategy for φ^{abstr}. Specifically, a counterstrategy is the tuple $C_{\varphi^{abstr}} = (Q, Q_0, \mathcal{X}, \mathcal{Y}, \delta, \gamma_{\mathcal{X}}, \gamma_{\mathcal{Y}})$, where Q is the set of counterstrategy states; $Q_0 \subseteq Q$ is the set of initial counterstrategy states; $\mathcal{X}, \mathcal{Y}$ are sets of propositions in AP; $\delta : Q \times 2^{\mathcal{Y}} \rightarrow 2^{Q}$ is a transition relation returning the set of possible successor states given the current state and valuations of robot commands in $\mathcal{Y}$; $\gamma_{\mathcal{X}} : Q \rightarrow 2^{\mathcal{X}}$ is a labelling function mapping states to the set of environment propositions that are `True` for incoming transitions to that state; and $\gamma_{\mathcal{Y}} : Q \rightarrow 2^{\mathcal{Y}}$ is a labelling function mapping states to the set of system propositions that are `True` in that state. We compute $C_{\varphi^{abstr}}$ using the `slugs` synthesis tool [32].

To find the graph cuts in the counterstrategy graph that prevent the environment from impeding the system, we first define the following propositional representation of state $q \in Q$ as $\psi(q) = \psi_{\mathcal{X}}(q) \wedge \psi_{\mathcal{Y}}(q)$, where

$$\psi_{\mathcal{Y}}(q) = \bigwedge_{\pi \in \gamma_{\mathcal{Y}}(q)} \pi \wedge \bigwedge_{\pi \in \mathcal{Y} \setminus \gamma_{\mathcal{Y}}(q)} \neg\pi, \tag{4.13}$$

$$\psi_{\mathcal{X}}(q) = \bigwedge_{\pi \in \gamma_{\mathcal{X}}(q)} \pi \wedge \bigwedge_{\pi \in \mathcal{X} \setminus \gamma_{\mathcal{X}}(q)} \neg\pi. \tag{4.14}$$

Next, letting $\delta_{\mathcal{Y}}(p) = \{q \in Q | \exists \pi \in \mathcal{Y} : q \in \delta(p, \pi)\}$, the set of *cut transitions* S_{cuts} is computed as $S_{cuts} = \{(p, q) \in Q^2 \mid q \in \delta_{\mathcal{Y}}(p), \psi(p) \wedge \psi(q) \models \bigvee_{i \in [1, n_{robots}]} \bigcirc \theta_S^i\}$. S_{cuts} collects those transitions on which the environment has intervened (by setting deadlock) to prevent the system from reaching its goals.

Finally, the following safety assumptions are found:

$$\varphi_{rev}^{e} = \square \bigwedge_{(p,q) \in S_{cuts}} (\psi_{\mathcal{Y}}(p) \wedge \psi_{\mathcal{X}}(p) \implies \neg \bigcirc \psi_{\mathcal{X}}(q)) \tag{4.15}$$

If any of the conjuncts in (4.15) falsify the antecedant of φ (the environment assumptions), they are discarded. Then, set $[\varphi_t^e]^{rev} = \varphi_t^e \wedge \varphi_{rev}^e$ and construct the final *revised* specification $\varphi^{rev} = \varphi_i^e \wedge [\varphi_t^e]^{rev} \wedge \varphi_g^e \implies [\varphi_i^s]' \wedge [\varphi_t^s]' \wedge \varphi_g^s$.

Algorithm 4.1 expresses our proposed approach for resolving deadlock. The automatically generated assumptions act to restrict the behavior of the dynamic obstacles. Each revision of the high-level specification excludes at least one environment move in a given state. Letting $|\cdot|$ denote set cardinality, with $2^{|\mathcal{X}|}$ environment actions and $2^{|\mathcal{Y}|}$ states, at most $2^{(|\mathcal{Y}|+|\mathcal{X}|)}$ iterations occur, though in our experience far fewer are needed. The generated assumptions are minimally restrictive – omitting even one allows the environment to cause deadlock, resulting in unrealizability. Note that the parsing step in line 7 creates statements that are displayed to the user. The user display step is explained in detail in the implementation in Section 4.9.

Algorithm 4.1: Find realizable φ^{rev} fulfilling task φ and resolving deadlock.

1: $\varphi^{abstr} \leftarrow \varphi_i^e \wedge \varphi_t^e \wedge \varphi_g^e \implies [\varphi_i^s]' \wedge [\varphi_t^s]' \wedge \varphi_g^s$
2: $[\varphi_t^e]^{rev} \leftarrow \varphi_t^e$; $\varphi^{rev} \leftarrow \varphi_i^e \wedge [\varphi_t^e]^{rev} \wedge \varphi_g^e \Rightarrow [\varphi_i^s]' \wedge [\varphi_t^s]' \wedge \varphi_g^s$
3: **while** φ^{rev} is *unrealizable* **do**
4: Extract $C_{\varphi^{rev}}$ from φ^{rev}
5: $\varphi_{rev}^e \leftarrow$ Eq. (4.15)
6: **for** each kth conjunct of φ_{rev}^e s.t. $\varphi_{rev}^e[k] \wedge [\varphi_t^e]^{rev} \neq$ `False` **do**
7: Parse $\varphi_{rev}^e[k]$ into human-readable statements and display to user.
8: $[\varphi_t^e]^{rev} \leftarrow [\varphi_t^e]^{rev} \wedge \varphi_{rev}^e[k]$
9: **end for**
10: $\varphi^{rev} \leftarrow \varphi_i^e \wedge [\varphi_t^e]^{rev} \wedge \varphi_g^e \Rightarrow [\varphi_i^s]' \wedge [\varphi_t^s]' \wedge \varphi_g^s$
11: **end while**

In practice, many of the added environment safety statements can be violated by dynamic obstacles at runtime without consequence, if these violations can be assumed to be temporary. For this reason, we introduce a recovery scheme that synthesizes a strategy that allows environment *safety* assumption violations to be tolerated. We refer the reader to [90] for these technical details of the details of this strategy. Note that we modify the approach to attempt a recovery only for violations of the newly added assumption φ^e_{rev}, rather than for the entire formula $[\varphi^e_t]'$, since our goal is to only make assertions on the environment's behavior with respect to deadlock and not all behaviors in general. The requirement for temporary deadlock is less restrictive than the requirement that deadlocks should *never* occur, but it nonetheless places additional requirements on the environment's behaviors, i.e. that the dynamic obstacles cannot infinitely often cause deadlock. Hence such conditions are displayed to the user in an easily-interpretable form.

Runtime certificates are displayed to the user in a format such as: `The task is guaranteed or robot 1 as long as any singleton deadlock occurring in the kitchen while heading to the door is eventually resolved on its own.` In this specific case, dynamic obstacles may enter deadlock with robot 1, but the obstacles are obligated to eventually resolve deadlock. If the dynamic obstacle is a person, the certificate may have no impact on the true behavior of the environment, as social norms deem it natural for people to resolve deadlocks on their own. If the dynamic obstacle is a door, then the certificate could alert that the door should eventually be opened to allow the robot to pass through. On the other hand, if the door never opens, then the certificate could help to explain that the door being closed as the reason the task remains unfulfilled.

It is possible that many such certificates are required, which may overwhelm the user. We address this in two ways. First, we project the found certificates onto the set of propositions relating to motion only, eliminating any propositions that do not relate to motion. Second, we use a graphical visualization of the certificates overlaid on a map of the physical workspace. In addition to the above provisions, the work in [21] offers an approach that can be adopted to further reduce the number of revisions fed to the user. There, a method is introduced for grouping regions that share the same properties for the revisions, and convey to the user metric information that is necessary for fulfilling the added revisions. Such an integration is left for future work. We refer the reader to Section 4.9 for implementation details.

4.6.2 Coordination Between Robots

Since the strategy for the robots' motion is completely determined at synthesis time, the controllers we synthesize should not lead to deadlocks if they can be safely avoided. For instance, two robots on the team should not enter a narrow doorway from opposite ends, only to become deadlocked there. This motivates the creation of a method for automatically inserting dimension-related information into the specification based on the workspace geometry and the volume of the robot so that the robots can pre-coordinate, at synthesis time, to avoid unneeded deadlock. This pre-coordination serves two purposes: 1) it allows to eliminate any environment assumptions between two robots in a region where there is high likelihood of deadlock if both are occupying that region, and 2) it changes the behavior of the agents to actively avoid potential deadlock in such high-risk regions, such as one-way corridors.

The modification considers the restrictions on what robots are allowed to do in certain regions, based on the dimension of the region and the size of the robot. We introduce an encoding of LTL formulas that eliminate the actions of robots that would result in deadlock. Specifically, we consider two cases: 1) a robot will not enter a region if the move will exceed the region's capacity and, 2) it will be prevented that two or more robots enter through opposite sides a one-way narrow region. We then create a new specification $\varphi^{abstr,coord}$ with pre-coordination of robots, and apply Algorithm 4.1 on $\varphi^{abstr,coord}$ by swapping out φ^{abstr} in line 1.

To create the LTL encoding, we introduce Algorithm 4.2 to enforce pairwise coordination amongst robots in the controlled team. If the region is too small to contain a pair of robots, any robot outside of the region is prevented from entering (line 5). If the boundary between two regions R_α and R_β is too small for two robots to pass through at once, and one robot is approaching the boundary from R_α (resp. R_β), then no other robot may approach that boundary if in R_β (resp. R_α). This requirement is encoded in lines 9–10. Note that Algorithm 4.2 is general to any workspace with convex regions.

4.7 On-line Local Motion Planning

In this section we describe the local planner that links the mission plan with the physical robot (recall Figure 4.5). At each step of the on-line execution, the synthesized strategy automaton provides a desired goal position for each robot and a preferred velocity $\bar{\mathbf{u}} \in \mathbb{R}^n$ towards it. An overview of the algorithm is given in Algorithm 4.3 and each step is described in detail in the following sections.

Algorithm 4.2: Augmenting a specification with agent coordination with respect to region geometry.

$D \leftarrow$ max dimension of the enclosing hull of the robots on the team
for each $R_\alpha \in \mathcal{R}$ **do**
  $A \leftarrow$ area of region R_α
  if $\frac{A}{D} < 1$ **then** ▷ *Region capacity is too small*
    $\varphi_t^s \leftarrow \varphi_t^s \wedge (\bigcirc \pi_\alpha^i \implies \neg \bigcirc \pi_{act,\alpha}^j) \wedge (\bigcirc \pi_\alpha^j \implies \neg \bigcirc \pi_{act,\alpha}^i)$
  end if
  for each $R_\beta \in Adj(R_\alpha)$ **do**
    if $\|R_\alpha \cap R_\beta\| < 2D$ **then** ▷ *Boundary between R_α and R_β is too narrow*
      $\varphi_t^s \leftarrow \varphi_t^s \wedge \bigwedge_{i,j=1}^{n_{robots}} \left((\psi_{\alpha\beta}^i \wedge \bigcirc \pi_\beta^j) \implies \bigcirc \neg \pi_{act,\alpha}^j \right)$
      $\varphi_t^s \leftarrow \varphi_t^s \wedge \bigwedge_{i,j=1}^{n_{robots}} \left((\psi_{\beta\alpha}^i \wedge \bigcirc \pi_\alpha^j) \implies \bigcirc \neg \pi_{act,\beta}^j \right)$
    end if
  end for
end for

We note that the reader may choose any other method for online planning as long as it preserves the avoidance guarantees with the kinematic model of the robots.

4.7.1 Local Motion Planning Overview

We build on the work on distributed Reciprocal Velocity Obstacles with motion constraints [3], and its recent extension to aerial vehicles [4].

As described by [3], the method follows two ideas. (a) The radius of the robot is enlarged by a pre-defined and typically fixed value $\varepsilon > 0$ for collision avoidance. This value depends on the kinodynamic model of the robot and

can be reduced in real time without having to recompute the stored maximum tracking errors. And, (b) in run time, the local trajectories are limited to those with a tracking error below ε with respect to their reference trajectory. Recall that the tracking errors where precomputed in the off-line process.

At each time-step an optimal reference velocity $\mathbf{u}^* \in \mathbb{R}^n$ is obtained by solving a convex optimization in reference velocity space. The associated local trajectory is guaranteed to be collision-free, satisfies the motion constraints and minimizes a cost function. The cost function minimizes the deviation to a pre-

---

Algorithm 4.3: Execution of the local planner using the synthesized strategy automaton.

1: **Input:** Current state of the robot, a local map, position and velocity of neighbors and a synthesized strategy automaton (FSM).
2: At each time instance (~ 10 Hz) do the following:
3: **if** the robot is in deadlock with any other agent **then**
4: Send a deadlock flag to the FSM.
5: **end if**
6: Obtain command from the FSM (e.g. "stay in the current room" or "move to the next room"), based on current state and deadlock flag.
7: Convert command into a goal position and preferred velocity $\bar{\mathbf{u}}$.
8: Compute constraints to satisfy the dynamic model of the robot.
9: **for** each neighboring agent **do**
10: Compute pairwise collision avoidance constraint.
11: **end for**
12: Compute largest obstacle-free convex region wrt static obstacles.
13: Solve constrained optimization to obtain collision-free motion
14: **Output:** A collision-free motion for the robot and the time horizon

---

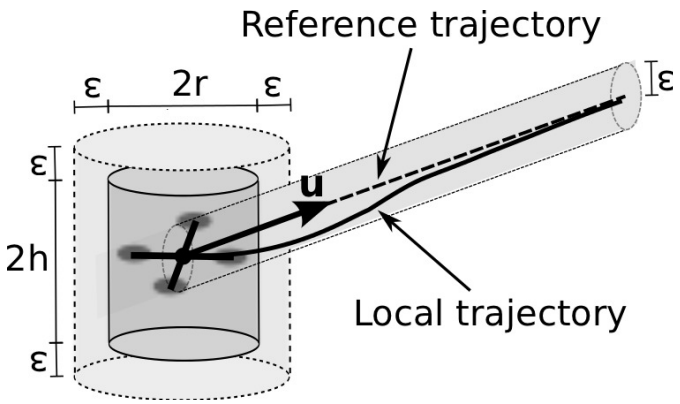


Figure 4.7: Schema local and reference trajectories for an aerial vehicle, generated from the reference velocity $\mathbf{u}$. The tracking error is limited by ε and the robot volume dilated by ε.

ferred velocity $\bar{\mathbf{u}}$, corrected by a small repulsive velocity $\mathring{\mathbf{u}}$ inversely proportional to the distance to the neighboring obstacles when in close proximity. As described by [4] this additional term introduces a desired separation between robots and obstacles. Note that the avoidance guarantees arise from the constrained optimization and not from the repulsive velocity.

4.7.2 Constraints

To define the motion and inter-agent avoidance constraints we build on the approach in [4]. We additionally introduce constraints for avoiding static obstacles. For completeness, we give an overview of each of the constraints.

Robot dynamics

Recalling (4.1) the motion constraint is given by the reference velocities for which the tracking error is below ε,

$$R(\mathbf{z}, \varepsilon) = \{\mathbf{u} \,|\, \gamma(\mathbf{z}, \mathbf{u}) \leq \varepsilon\}. \tag{4.15}$$

approximated by the largest inscribed convex polytope/ellipsoid $\hat{R}(\mathbf{z}, \varepsilon) \subset R(\mathbf{z}, \varepsilon)$.

Avoidance of other agents

Denote by $\mathbf{p}_j$, $\mathbf{v}_j$, $\bar{r}_j$ and $\bar{h}_j$ the position, velocity, dilated radius and height of a neighboring agent j. Assume that it keeps its velocity constant for $\tilde{t} \leq \tau$. Reciprocity (i.e. the other agent follows the same algorithm) can as well be assumed and is discussed in [4]). For every neighboring agent j, the constraint is given by the reference velocities $\mathbf{u}$ for which the agents' enveloping shape do not intersect within the time horizon. For cylindrically-shaped agents moving in 3D the velocity obstacle of colliding velocities is a truncated cone

$$\begin{aligned} VO_j^\tau = \{\mathbf{u} \,|\, \exists \tilde{t} \in [0, \tau] : \|\mathbf{p}^H - \mathbf{p}_j^H + (\mathbf{u}^H - \mathbf{v}_j^H)\tilde{t}\| \leq \bar{r} + \bar{r}_j \\ \text{and } |p^V - p_j^V + (u^V - u_j^V)\tilde{t}| \leq \bar{h} + \bar{h}_j\}, \end{aligned}$$

where $\mathbf{p} = [\mathbf{p}^H, p^V]$, with $\mathbf{p}^H \in \mathbb{R}^2$ its projection onto the horizontal plane and $p^V \in \mathbb{R}$ its vertical component. The constraint is linearized to $A_j(\mathbf{p}, \varepsilon) = \{\mathbf{u} \,|\, \mathbf{n}_j^T \mathbf{u} \leq b_j\}$, where $\mathbf{n}_j \in \mathbb{R}^3$ and $b_j \in \mathbb{R}$ maximize $\mathbf{n}_j^T \mathbf{v} - b_j$ subject to $A_j(\mathbf{p}, \varepsilon) \cap VO_j^\tau = \emptyset$.

Avoidance of static obstacles

We extend a recent fast iterative method to compute the largest convex polytope in free space [26], by directing the growth of the region in the preferred direction of motion and enforcing that both the current position of the robot and a look ahead point in the preferred direction of motion are within the region. The convex polytope is computed in position space ($\mathbb{R}^3$ for aerial vehicles) and then converted to an equivalent region in reference velocity space. See Algorithm 4.4, where $directedEllipsoid(\mathbf{p}, \mathbf{q})$ is the ellipsoid with one axis given by the segment $\mathbf{p} - \mathbf{q}$ and the remaining axis infinitesimally small, and K the number of steps in the linear search, typically between 2 and 4.

Algorithm 4.4: Largest collision-free directed convex polytope.

$L \leftarrow \mathbf{p} + \bar{\mathbf{u}}\{\tau, \frac{K-1}{K}\tau, \frac{K-2}{K}\tau, \ldots, 0\}$; $\quad P := \emptyset$;
$\mathbf{q} \leftarrow L[0]$; $\quad L := L \setminus \mathbf{q}$;
while $L \neq \emptyset$ and $\mathbf{p}, \mathbf{q} \notin P$ **do**
  $E \leftarrow directedEllipsoid(\mathbf{p}, \mathbf{q})$ ▷ *Largest polytope seeded in E computed as in [26]*
  while not converged **do**
    $P \leftarrow$ separating planes of E and dilated O (QP)
      such that $P \subset \mathbb{R}^n \setminus (O + V_\varepsilon)$
    If $\mathbf{p}, \mathbf{q} \notin P$ **then** { $\mathbf{q} \leftarrow L[0]$; $L := L \setminus \mathbf{q}$; **break**; }
    $E \leftarrow$ ellipsoid $E \subset P$ of maximal volume (SDP)
  end while
end while
$F(\mathbf{p}, \varepsilon) := (P - \mathbf{p})/\tau$ ▷ *Converts to ref. velocity, $\boldsymbol{u}$, space*

Avoiding incorrect region transitions

The local planner prevents incorrect region transitions (for instance, avoiding entering another region if the robot's local goal is within the current one) by introducing "virtual" doors at borders between workspace regions. These virtual doors may be closed or opened depending on the desired transition. A closed door is introduced as an obstacle in O.

4.7.3 Optimization

The optimization cost is given by two parts. As described in Section 4.2.3, the first one is a regularizing term, weighted by a design constant $\bar{\alpha}$, and the second one is a minimizer with respect to a preferred velocity.

A convex optimization with quadratic cost and linear and quadratic constraints is solved

$$\begin{aligned} \mathbf{u}^* := \arg\min_{\mathbf{u}\in\mathbb{R}^n} \quad & (\alpha\|\mathbf{u}-\mathbf{v}\|^2 + \|\mathbf{u}-(\bar{\mathbf{u}}+\mathring{\mathbf{u}})\|^2), \\ \text{s.t.} \quad & \mathbf{u} \in \hat{R}(\mathbf{z},\varepsilon) \cap F(\mathbf{p},\varepsilon) \\ & \mathbf{u} \in A_j(\mathbf{p},\varepsilon) \quad \forall j \text{ neighbor agent} \end{aligned} \tag{4.14}$$

The solution of this optimization is a collision-free reference velocity $\mathbf{u}^*$ which minimizes the deviation towards the goal specified by the strategy automaton. The associated trajectory (see Section 4.2.3) is followed by the robot and is collision-free.

4.7.4 Deadlock Detection

To allow the strategy automaton to resolve deadlock at runtime, we set the deadlock proposition x^{ij} ($i = 1, \ldots, n_{robots}$, $j = 0, \ldots, n_{robots}$; $j = 0$ implying a dynamic obstacle), according to the following rule:

$$x^{ij} \Leftarrow (\|\mathbf{u}_i^*\| < k_1) \wedge (\|\bar{\mathbf{u}}_i\| > k_2) \wedge (\|\mathbf{p}_i - \mathbf{p}_j\| < k_3), \tag{4.15}$$

with $k_1, k_2, k_3 > 0$ being tunable parameters. This states that a necessary condition for x^{ij} to be set is when the agent velocity magnitude $\|\mathbf{u}_i^*\|$ is low, the preferred velocity magnitude $\|\bar{\mathbf{u}}_i\|$ is high, and the unsigned distance between agents $\|\mathbf{p}_i - \mathbf{p}_j\|$ is within a prescribed tolerance. In our experiments these values are chosen experimentally to detect all deadlocks while minimizing false positives. We introduce a small hysteresis in the flag activation. In particular, we activate the deadlock flag when the right-hand condition of (4.15) has been `True` for a minimum period of time $T_{dk-true}$. When the flag becames active, x^{ij} is kept in `True` for a minimum period of time $T_{dk-false}$. In our experiments we employ $8s$ and $5s$ respectively. This hysteresis prevents false alarms and chattering when the velocity is small (e.g. while the robot is accelerating).

4.8 Theoretical Guarantees

We provide proofs for the guarantees inherent to our synthesized controller. The following three subsections are sufficient to show that, under the collision-free guarantees provided by the local planner, the synthesized strategy realizes the reactive task specification and resolves deadlocks.

4.8.1 Correctness With Respect to Robot Dynamics

By construction of the local planner, the controller is guaranteed correct with respect to the low-level controller $f(\mathbf{z}, \mathbf{u}, \tilde{t})$, which is continuous on the initial state of the robot and respects its dynamics. We do assume that the model of the robot is accurate and that there are no external disturbances.

4.8.2 Collision-Free Motion

We claim that local planner of Section 4.7 yields collision-free motion in dynamic environments, under the constant velocity assumption. A proof of this fact is contained in Theorem 1 of [2]; however, a sketch of the proof is as follows.

If (4.7.3) is *feasible*, collision-free motion is guaranteed for the local trajectory up to time τ with the assumption that all interacting agents maintain a constant velocity. The foundation of the proof is included in the work of [4], with extension to treat the case of a dynamic obstacle maintaining a constant velocity, in which all agents are always assumed to perform reciprocal collision avoidance.

On the other hand, if (4.7.3) is *infeasible*, no collision-free solution exists that respects all of the constraints. If the time horizon τ of the local planner is larger than the required time to stop - typically a couple of seconds - passive safety is preserved by slowing down on the last feasible path and eventually reaching a stop. Also, since this computation is performed at a high frequency, each individual robot is able to adapt to changing situations, and the resulting motion is collision-free.

4.8.3 Correctness with Respect to the Task Specification

Since the local planner is myopic, it provides guarantees up to a time horizon τ and consequently may result in deadlock and livelock. However, as we have shown, the planner's local guarantees allow a discrete abstraction that the strategy automaton can use to resolve deadlocks and avoid livelocks. Here we formally prove the guarantees on the execution provided by our synergistic on-line and offline synthesis.

Proposition 4.1. *Given a task specification φ that ignores collisions, if the resulting specification φ^{abstr} defined in Section 4.5 is realizable, then the corresponding strategy automaton also realizes φ.*

Proof. Assume given $\varphi = \varphi_i^e \wedge \varphi_t^e \wedge \varphi_g^e \implies \varphi_i^s \wedge \varphi_t^s \wedge \varphi_g^s$. Recall that $\varphi^{abstr} = \varphi_i^e \wedge \varphi_t^e \wedge \varphi_g^e \implies [\varphi_i^s]' \wedge [\varphi_t^s]' \wedge \varphi_g^s$, where $[\varphi_i^s]'$ and $[\varphi_t^s]'$ contain φ_i^s and φ_t^s as subformulas, respectively. Suppose that strategy automaton $\mathcal{A}_{\varphi^{abstr}}$ realizes φ^{abstr}. This means that the resulting controller is guaranteed to fulfill the requirement $[\varphi_i^s]' \wedge [\varphi_i^s]' \wedge \varphi_g^s$ as long as the environment fulfills the assumption $\varphi_i^e \wedge \varphi_t^e \wedge \varphi_g^e$. This implies that $A_{\varphi^{abstr}}$ fulfills $\varphi_i^s \wedge \varphi_t^s \wedge \varphi_g^s$ as long as the environment fulfills the assumption $\varphi_i^e \wedge \varphi_t^e \wedge \varphi_g^e$. □

Proposition 4.2. *Given a task specification φ that ignores collisions, if φ is realizable but the resulting specification φ^{abstr} is not realizable, then the revision procedure in Section 4.6.1 will find an assumption φ_{rev}^e to add to φ^{abstr} that renders the resulting specification φ^{rev} realizable and the resulting strategy $\mathcal{A}_{\varphi^{rev}}$ free of deadlock and livelock.*

Proof. Suppose φ is realizable by strategy $\mathcal{A}_\varphi$, but φ^{abstr} is not realizable, admitting counterstrategy $C_{\varphi^{abstr}} = (Q, \ldots)$. It suffices to show that the set S_{cuts} is

nonempty. Assume by way of contradiction that S_{cuts} is empty. Then the rising edge of deadlock θ_s^i never occurs for any i, so no robot transitions are ever disabled. Since we assume that deadlock does not occur in the initial state, this means that x^{ij} is always `False` for every i,j. Therefore $[\varphi_i^s]' \wedge [\varphi_t^s]' \wedge \varphi_g^s$ defined in Section 4.5 reduces to $\varphi_i^s \wedge \varphi_t^s \wedge \varphi_g^s$. The lack of deadlock means that any region transition contained in $\mathcal{A}_\varphi$ is still admissible, and therefore $\mathcal{A}_\varphi$ can be used as a strategy to realize φ^{abstr}, a contradiction. Therefore, there must be deadlock and S_{cuts} is not empty. Now, upon addition of the assumptions φ^{abstr}, existence of $\mathcal{A}_{\varphi^{rev}}$ that satisfies φ^{abstr} implies, by construction, that $\mathcal{A}_{\varphi^{rev}}$ is livelock-free. □

Note that it may be the case that S_{cut} is nonempty, but for every $(p,q) \in S_{cuts}$, the resulting revision

$$(\psi_{\mathcal{Y}}(p) \wedge \psi_{\mathcal{X}}(p) \implies \neg \bigcirc \psi_{\mathcal{X}}(q))$$

contradicts φ_e^t. This indicates that φ is only realizable because it makes unreasonable assumptions on the environment. Our approach identifies this fact as a by-product of the revision process.

4.8.4 Computational Complexity

For a given choice of m, the offline reactive synthesis algorithm used in this work is exponential in the number of propositions [11, 32]. Using our encoding, the problem scales linearly with n_{robots} – no worse than existing approaches (e.g. [86]). When one or more dynamic obstacles are considered, the number of propositions does not change. As stated in Section 4.6, $2^{(|\mathcal{Y}|+|\mathcal{X}|)}$ iterations of the main loop in Algorithm 4.1 are needed in the worst case, yielding a theoretical complexity that is doubly exponential in the number of propositions.

For the on-line component, a convex program is solved independently for each robot, with the number of constraints linear in the number of neighboring robots. The runtime of the iterative computation of the convex volume in free space barely changes with the number of obstacles, up to tens of thousands [26], and a timeout can be set, with the algorithm returning the best solution found.

4.9 Experiments and Simulations

We present results of our end-to-end approach both in simulation and on hardware. Our evaluation is meant to illustrate the various parts of the synthesis and execution process, and provide a statistically-grounded evaluation of the approach when placed in a difficult environment that does not necessarily behave according to the automatically-generated environment assumptions. In this context, our results reveal that our approach has merit in dealing with such environments to execute the task successfully. We furthermore show that our approach is scalable to any number of dynamic obstacles, and that the local planner applies to 3-D workspaces. Lastly, we show that our approach may be executed in real time on actual hardware.

The synthesis procedure described in Section 4.5 was implemented with the `slugs` synthesis tool [32], and executed with the LTLMoP toolkit [37]. The local motion planner, Section 4.7, was implemented with the IRIS toolbox [26] and an off-the-shelf convex optimizer. We assume the dynamic obstacles are cooperative in avoiding collisions, therefore, each one is controlled by a local planner. Many of the experiments presented in this section are available in the accompanying video.

In what follows, we consider a "garbage collection" scenario, upon which we synthesize a strategy automaton.

Example 4.2 (Garbage Collection). *A robot team is required to patrol the Living Room (R_{LR}) and Bedroom (R_{BR}) of the workspace in Figure 4.8. For two robots, the specification is:*

$$\square\diamondsuit(\pi^1_{LR}) \wedge \square\diamondsuit(\pi^1_{BR}) \wedge \square\diamondsuit(\pi^2_{LR}) \wedge \square\diamondsuit(\pi^2_{BR})$$

and if garbage is observed, pick it up

$$\square(\pi^1_{garb} \implies \pi^1_{act,pickup}) \wedge \square(\pi^2_{garb} \implies \pi^2_{act,pickup}).$$

Additionally, the robots must always avoid other moving agents.

The system propositions are actions to move between regions ($\pi^i_{act,LR}, \ldots, \pi^i_{act,BR}$) and to pick up ($\pi^i_{act,pickup}$). The environment propositions are sensed garbage (π^i_{garb}), region completions ($\pi^i_{LR}, \ldots, \pi^i_{BR}$), and pick up completion (π^i_{pickup}).

We omit the complete encoding of Def. 4.2, however, for illustration we supply the transition formulas for the case where robot 1 is in Hall:

$$\varphi^s_t : \begin{cases} \square(\pi^1_{Hall} \vee \pi^1_{LR} \vee \pi^1_{BR} \vee \pi^1_{Kitchen} \vee \pi^1_{Door}) \\ \square(\bigcirc \pi^1_{Hall} \Longrightarrow \bigcirc \pi^1_{act,Hall} \vee \bigcirc \pi^1_{act,BR} \vee \bigcirc \pi^1_{act,LR}) \end{cases}$$

$$\varphi^e_t : \begin{cases} \square(\pi^1_{act,Hall} \vee \pi^1_{act,LR} \vee \pi^1_{act,BR} \vee \pi^1_{act,Kitchen} \vee \pi^1_{act,Door}) \\ \square(\pi^1_{Hall} \wedge \pi^1_{act,Hall} \Longrightarrow \bigcirc \pi^1_{Hall}) \\ \square(\pi^1_{Hall} \wedge \pi^1_{act,LR} \Longrightarrow \bigcirc \pi^1_{Hall} \vee \bigcirc \pi^1_{LR}) \\ \square(\pi^1_{Hall} \wedge \pi^1_{act,BR} \Longrightarrow \bigcirc \pi^1_{Hall} \vee \bigcirc \pi^1_{BR}) \end{cases}$$

The initial conditions φ^s_i and φ^e_i are `True`.

We implement the above example using humanoid robots (able to rotate in place, move forward and along a curve) and simulated quadrotor UAVs.

4.9.1 Synthesis and Revisions

Upon synthesizing a controller for single robot, we obtain 16 revisions to the environment assumptions. These are displayed to the user as runtime certificates. One example is: `Deadlock should not occur when the robot is in the Hall moving toward the Living Room and had already been blocked from entering the Bedroom`. Note that, with each robot added to the team, the number of revisions grows combinatorially. In contrast to the single-robot case, there are a total of 1306 statements given to the user in the case of two robots. In these cases, we display the revisions graphically, by projecting over the variables of interest: the current region and action for each robot. Rather than displaying all 1306 statements, we show the projection consisting of 45 statements projected onto the set of each robot's motion and activation propositions. Satisfying these 45 statements implies that we also satisfy the 1306 statements. To further aid the user, we display them graphically on the workspace as shown in Figure 4.8.

For instance, a red arrow on the boundary of the Hall indicates that the automaton cannot guarantee the task if the robot experiences deadlock when it is in the Hall and while activating a motion to the Living Room. The certificates displayed in Figure 4.8 are projections onto a subset of the complete set of propositions (i.e. deadlocks, memory propositions, robot positions, and robot actions for *each* of the robots in the team), by abstracting those variables away. That is, if there exist restrictions on deadlock for *any* of the propositions that have been abstracted away, then the revision displayed will be a conservative overapproximation to the true revision and the dot will be labeled red.

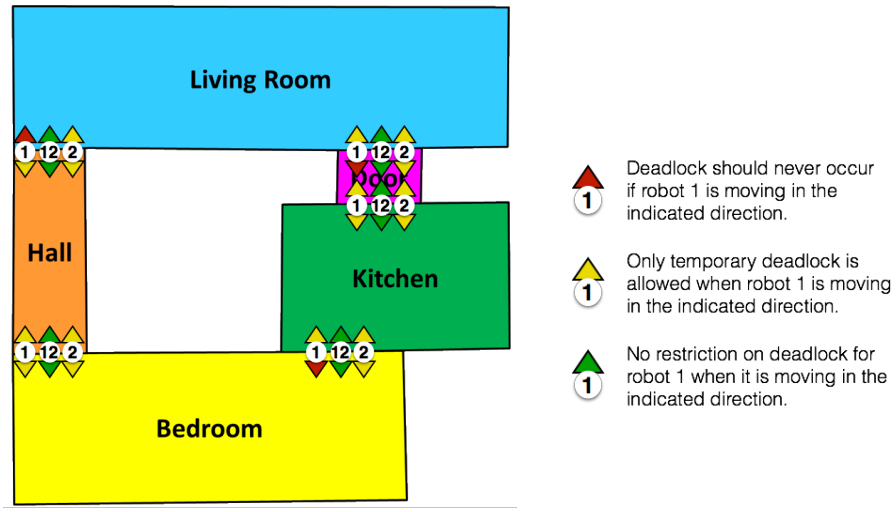


Figure 4.8: Workspace showing specification revisions for each region completion/activation pairs where singleton or pairwise deadlock may occur. An arrow's color indicates the type of assumption that has been made. The number (or pair of numbers) indicates the robot (or robot pairs) concerned with the assumption. The placement of the arrow indicates the region that the robot is headed (i.e. its action commands $AP_{\mathcal{R}}^{act}$) when the given assumption holds true.

4.9.2 Scalability with Respect to Dynamic Obstacles

Considering Example 4.2, the specification for the single-robot case consists of 14 propositions, while that for the two-robot case consists of 29 propositions. The specification is invariant to the number of dynamic obstacles in either case.

One could also consider a two-robot team controlled by a baseline strategy that relies on mutual exclusion (one robot per region) to be kept with other robots and dynamic obstacles (DO). That strategy required 20 propositions for the case without DOs. One additional proposition is added for each region for each DO (producing 25 for one DO, 35 for three DO, 60 propositions for eight DO, etc.). Because the obstacles are assumed to behave in an adversarial manner, they can violate mutual exclusion if they enter a neighboring region of the

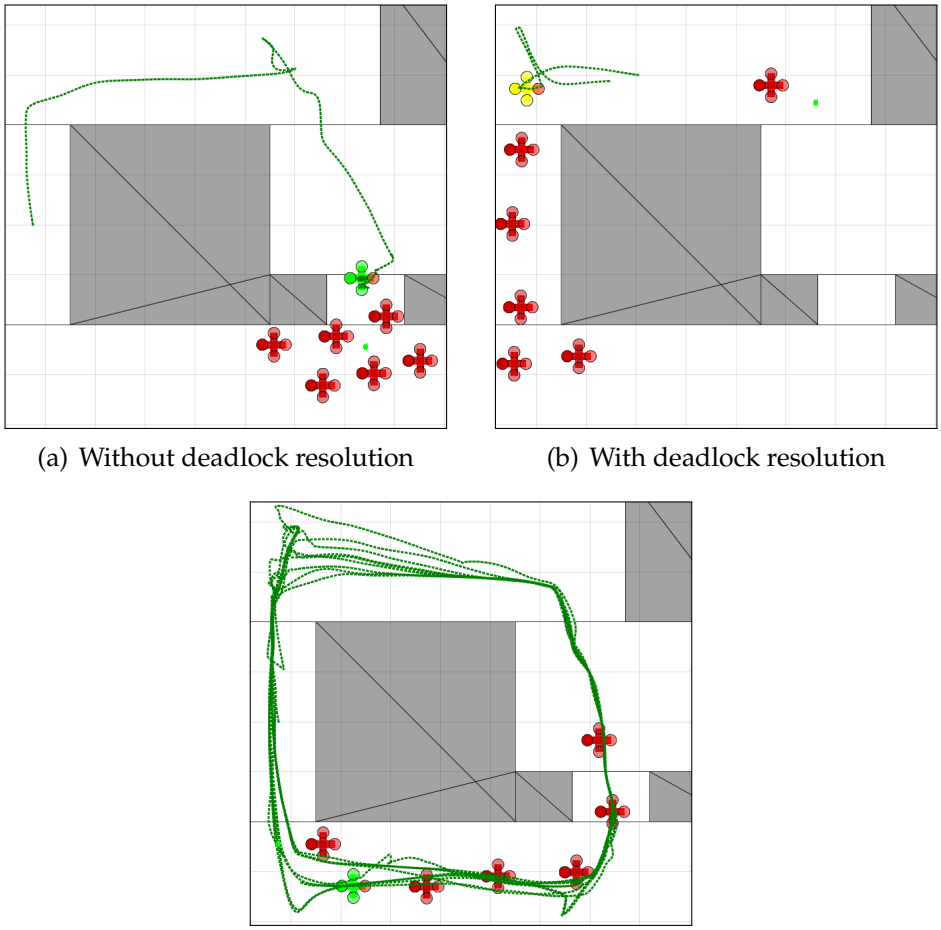
robot. Hence, the baseline synthesis procedure is *not realizable* for one or more dynamic obstacles.

In contrast, our approach is realizable independently of the number of dynamic obstacles and requires fewer propositions than the case with two or more DO.

4.9.3 Performance Evaluation

We directly compare the proposed approach with a baseline approach where the robots execute a local planner, but there is no deadlock resolution in the strategy. Recall that there is no guarantee of mission satisfaction in that case. Figures 4.9, 4.10 and 4.11 display results for various problem scenarios. In each experiment, we use the model described in [4] to model the robots and dynamic obstacles as quadrotors. The "counter-flow" cases follow a pre-defined set of waypoints that allow DOs to circulate within the workspace in one direction (counter to the flow of the robots), while, in the "random waypoints" cases, DOs randomly select a neighboring waypoint once a waypoint has been achieved. To detect deadlock, we use the criteria in (4.15) with the choice of parameters $k_1 = \frac{1}{3}$, $k_2 = \frac{1}{4}$, and $k_3 = 1.5$.

Each test case consisted of 133 minutes of data obtained over multiple simulation runs lasting 200 seconds each. The simulation was terminated before 200 seconds if none of the controlled robots reached their goal, but none had been moving (their velocity falls below a threshold) for 100 seconds or longer. Any such runs were flagged as *unresolved deadlock*, at which point the robots are deemed unable to continue their task. The robots in the team were initialized



(a) Without deadlock resolution

(b) With deadlock resolution

(c) Path with deadlock resolution

Figure 4.9: Example of the approach in a scenario with six dynamic obstacles (dark red) and one controlled quadrotor (light green/yellow). The path of the controlled quadrotor is shown with a dashed green line. (a) The original approach without deadlock resolution avoids collisions but can get into unresolved deadlocks. The path leading to the deadlock is shown. (b) The proposed approach successfully resolves deadlocks, like the one shown here. The path leading to and resolving the deadlock are shown. (c) Path of the controlled quadrotor using the proposed approach during a ten minutes simulation. The quadrotor successfully avoids collisions and reverts the motion when it encounters a deadlock.

randomly at different regions in the workspace.

In the "counter-flow" example of Figure 4.11, 100% of the simulation runs without the proposed deadlock resolution approach eventually enter unresolved deadlock at some point during the run. In contrast, when the proposed approach is used, deadlock is able to be resolved, resulting in more goals being visited. In the single-robot case, only 5% of the runs lead to unresolved deadlock. In all such runs, the DOs had violated a runtime certificate (note that the DOs were not programmed to satisfy any such certificates); in some cases the DOs surrounded the robot. In the two-robot case, nearly 20% of the runs lead to unresolved deadlock. This number is higher than in the one-robot case because there is more than one robot whose motion could be blocked by the DOs, lead-

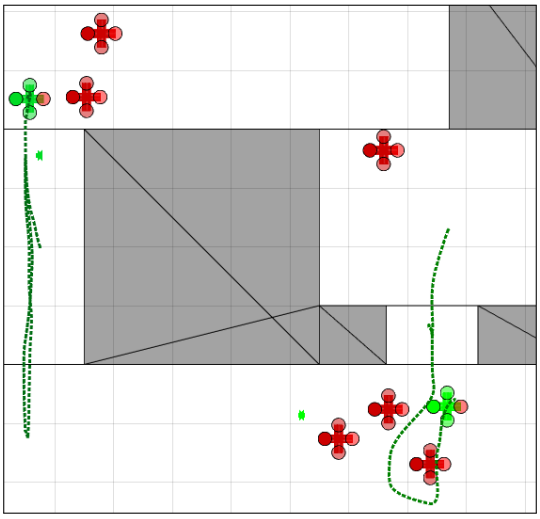
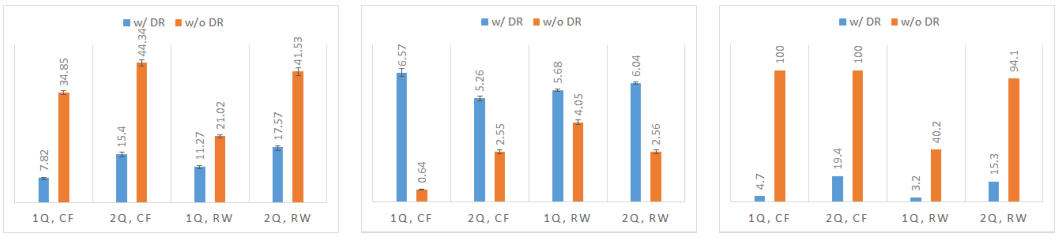


Figure 4.10: Example of the approach with six dynamic obstacles (dark red) and two controlled quadrotors (light green). The dynamic obstacles navigate to randomized locations and the controlled robots execute the proposed framework. The path of the controlled quadrotors is shown with a dashed green line for one minute of the simulation. The quadrotors successfully avoid collisions, reverse motion when they encounter a deadlock and explore the top and bottom rooms.



(a) # encountered deadlocks per 200-sec run (b) # goals visited per 200-sec run (c) # unresolvable deadlocks per 200-sec run

Figure 4.11: Comparison of the results of the "garbage collection" scenario with DOs exhibiting counter-flow (CF) or random waypoints (RW) behaviors, with either one quadrotor (1Q) or two quadrotors (2Q). For each scenario, we evaluate the results for data collected over 40 200-sec runs. Six quadrotors were used for the DOs. Over each run, (a), (b), and (c) show, respectively, data for the number of encountered deadlocks, the number of goals visited, and the number of unresolvable deadlocks. Standard deviations are indicated as error bars in (a) and (b).

ing to more encountered deadlocks. Additionally, when one robot has already become deadlocked, the objects in the environment effectively act as static obstacles to the remaining robot, increasing the chance it will become deadlocked as compared with moving, dynamic obstacles. The combined effect of these two factors is the reason why there is a four-fold increase in the number of encountered deadlocks.

The "random waypoints" cases are included to evaluate the performance of the proposed approach where the DOs do not all move in the same direction, but instead move randomly in the workspace. In the case of a single robot, deadlock resolution allows the robots to find alternate routes around deadlocks, and thus the robot is able to visit 40% more goals than the case without deadlock resolution. In the case of two robots, the team is able to achieve 136% more goals than without resolution. As may be observed in the supplementary videos, deadlock resolution gives the robots an ability to exploit areas of the workspace

containing a lower density of dynamic obstacles to achieve their goals. The cases where deadlock resolution is included results in greater likelihood of task achievement over a 200-second interval. As compared with the counter-flow cases, there are fewer cases of unresolved deadlock because the random nature of the DOs allows the robots to move more freely in some cases than in others.

4.9.4 3D Problem Domain

We next demonstrate the effectiveness of the approach in a 3D scenario where, in the $5\times5\times5$ m^3 two-floor workspace of Figure 4.12, robots move between floors through a vertical opening at the left corner or the stairs at the right side. The two robots on the team as well as the dynamic obstacle are simulated quadrotors. The task is to infinitely often visit the top and bottom floors while avoiding collisions and resolving deadlock. The strategy automaton is synthesized as described in Section 4.5. A local planner for the 3D environment is constructed following Section 4.7. A representative experiment is shown in the snapshots in Figure 4.12. The green robot enters deadlock when moving towards the upwards corridor; however, deadlock is resolved by taking the alternative route up the stairs.

4.9.5 Physical Experiments

To demonstrate effectiveness in a physical setting, we employ two Aldebaran Nao robots to carry out the planar garbage collection scenario, with a teleoperated KUKA youBot serving as the dynamic obstacle. The model for the Nao

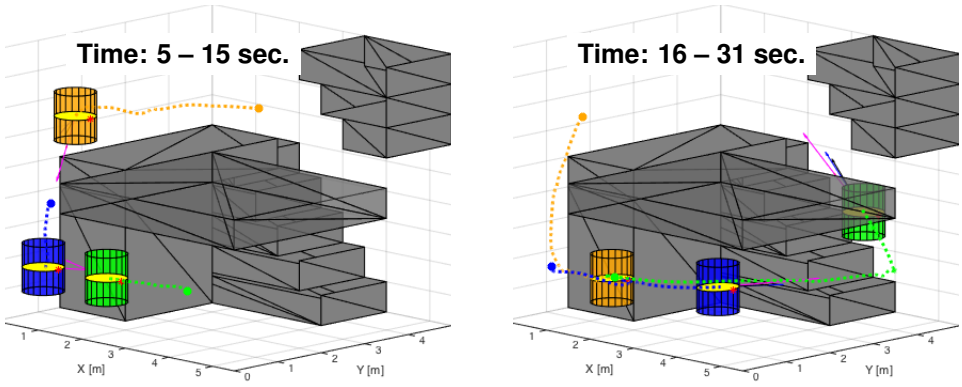
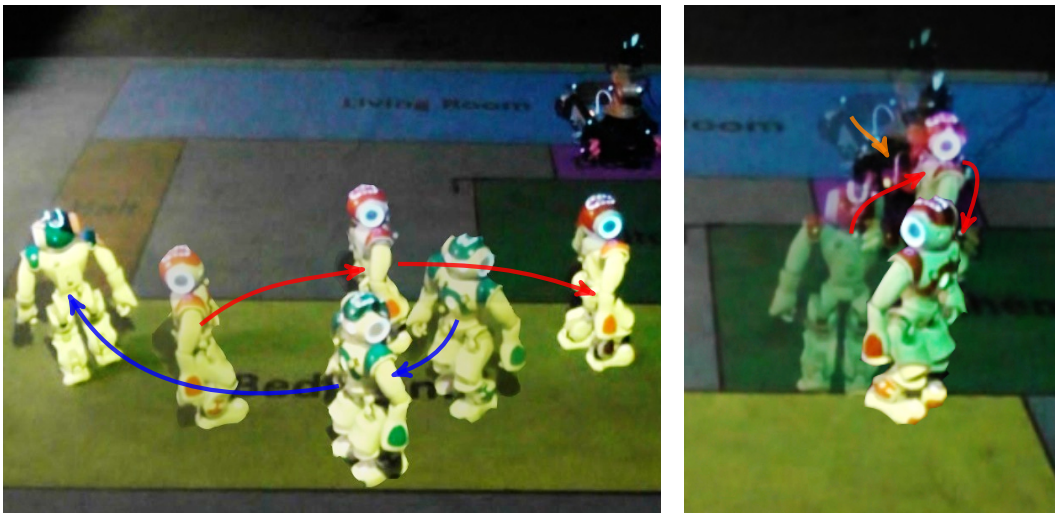


Figure 4.12: Deadlock resolution (green robot) and safe navigation in a 3D environment. Quadrotors are displayed at the final time and their paths for the time interval. Each yellow disk represents a quadrotor and the cylinder its safety volume. The orange robot represents the dynamic obstacle.

robots is one where the robots are are able to rotate in place, move forward, and move along a curve at a constant velocity. The size of the field is 5m by 3m, and the sensing range for the local planner is 1m. The size is such that only one Nao robot may fit through the Hall and Door at a time. The positions of each robot are measured through a motion capture system. The local planner is implemented on a laptop computer communicating via a WiFi connection to the robots. In the local planner, the Nao robots are taken to have a circular footprint with effective radius of 0.2 m.

We carried out experiments using two robots on the team, using the workspace shown in Fig. 4.8. The revisions for these two robots are pictured in the figure for the synthesized mission plan. As demonstrated in the snapshots in Fig. 4.13, the Naos can execute the task, by avoiding collisions and resolving deadlocks with one another and with the dynamic obstacle (the KUKA youBot). At the particular deadlock event shown in Fig 4.13(b), the youBot must eventually move away from the Door region, as the assumption pictured in Fig. 4.8



(a) Avoidance maneuver (b) Deadlock resolution

Figure 4.13: Planar scenario with two centrally-controlled Nao robots and a dynamic obstacle (youBot). In each image, three consecutive frames of the robot's motion are superimposed. In (a), the local planner enables the two Naos to avoid collisions with each other. In (b), one of the Naos reverses direction to resolve the deadlock with the youBot.

states that 'only temporary deadlock is allowed' when either of the robots are trying to enter it from the Kitchen. The experiments demonstrate that the approach is effective at deadlock resolution and at achieving collision free motion, thereby satisfying the mission specification.

4.10 Conclusion

We present a framework for synthesizing a strategy automaton and collision-free local planner that guarantees completion of a task specified in linear temporal logic, where we consider reactive mission specifications abstracted with respect to basic locomotion, sensing and actuation capabilities. Our approach

is less conservative than current approaches that impose a separation between agents, and is computationally cheaper than explicitly modeling all possible obstacles in the environment. If no controller is found that satisfies the specification, the approach automatically generates the needed assumptions on deadlock to render the specification realizable and communicates these to the user. The approach generates controllers that accommodate deadlock between robots or with dynamic obstacles *independently of* the precise number of obstacles present, and we have shown that the generated controllers are correct with respect to the original specification. Experiments with ground and aerial robots demonstrate collision avoidance with other agents and obstacles, satisfaction of a task, deadlock resolution and livelock-free motion. Future work includes optimizing the set of revisions found, and decentralizing the synthesized controller.

BIBLIOGRAPHY

[1] J Alonso-Mora, A Breitenmoser, M Rufli, P Beardsley, and R Siegwart. Optimal Reciprocal Collision Avoidance for Multiple Non-Holonomic Robots. *Proc. Int. Symp. on Distributed Autonomous Robotics Systems*, 2010.

[2] J. Alonso-Mora, J. A. DeCastro, V. Raman, D. Rus, and H. Kress-Gazit. Reactive mission and motion planning while avoiding dynamic obstacles. *in submission*, 2016.

[3] J Alonso-Mora, P Gohl, S Watson, R Siegwart, and P Beardsley. Shared control of autonomous vehicles based on velocity space optimization. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pages 1639–1645, 2014.

[4] J Alonso-Mora, T Naegeli, R Siegwart, and P Beardsley. Collision Avoidance for Multiple Aerial Vehicles. *Autonomous Robots*, 2015.

[5] Rajeev Alur, Salar Moarref, and Ufuk Topcu. Counter-strategy guided refinement of gr(1) temporal logic specifications. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 26–33, 2013.

[6] Ebru Aydin Gol, Mircea Lazar, and Calin Belta. Language-guided controller synthesis for discrete-time linear systems. In *Proceedings of the 15th ACM International Conference on Hybrid Systems: Computation and Control*, pages 95–104. ACM, 2012.

[7] Calin Belta and L.C.G.J.M. Habets. Constructing decidable hybrid systems with velocity bounds. In *Proc. of the 43rd IEEE Conf. on Decision and Control (CDC 2004)*, pages 467–472, Bahamas, 2004.

[8] José Bento, Nate Derbinsky, Javier Alonso-Mora, and Jonathan S Yedidia. A message-passing algorithm for multi-agent trajectory planning. *Annual Conference on Neural Information Processing Systems NIPS*, 2013.

[9] John T. Betts. *Practical Methods for Optimal Control and Estimation Using Nonlinear Programming*. Cambridge University Press, New York, NY, USA, 2nd edition, 2009.

[10] A. Bhatia, L. E. Kavraki, and M. Y. Vardi. Sampling-based motion planning with temporal goals. In *Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA 2010)*, pages 2689–2696, 2010.

[11] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive (1) designs. *Journal of Computer and System Sciences*, 78(3):911–938, 2012.

[12] Roderick Bloem, Alessandro Cimatti, Karin Greimel, Georg Hofferek, Robert Könighofer, Marco Roveri, Viktor Schuppan, and Richard Seeber. *RATSY – a new requirements analysis tool with synthesis*, pages 425–429. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[13] R.R. Burridge, A.A. Rizzi, and D.E. Koditschek. Sequential composition of dynamically dexterous robot behaviors. *The International Journal of Robotics Research*, 18:534–555, 1999.

[14] Yushan Chen, Xu Chu Ding, Alin Stefanescu, and Calin Belta. Formal approach to the deployment of distributed robotic teams. *IEEE Transactions on Robotics*, 28(1):158–171, 2012.

[15] Marcello Cirillo, Tansel Uras, and Sven Koenig. A lattice-based approach to multi-robot motion planning for non-holonomic vehicles. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Chicago, USA, 2014.

[16] Igor Cizelj and Calin Belta. Probabilistically safe control of noisy dubins vehicles. In *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS 2012)*, pages 2857–2862. IEEE, 2012.

[17] David C. Conner, Howie Choset, and Alfred A. Rizzi. Integrated planning and control for convex-bodied nonholonomic systems using local feedback control policies. In *Robotics: Science and Systems*, 2006.

[18] David C Conner, Alfred Rizzi, and Howie Choset. Composition of local potential functions for global robot control and navigation. In *Proc. of 2003 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS 2003)*, volume 4, pages 3546– 3551. IEEE, October 2003.

[19] J. A. DeCastro, R. Ehlers, M. Rungger, A. Balkan, P. Tabuada, and H. Kress-Gazit. Dynamics-based reactive synthesis and automated revisions for high-level robot control. *CoRR*, abs/1410.6375, 2014.

[20] J. A. DeCastro and H. Kress-Gazit. Synthesis of nonlinear continuous controllers for verifiably-correct high-level, reactive behaviors. *The International Journal of Robotics Research*, 34(3):378–394, 2015.

[21] Jonathan DeCastro, Rüdiger Ehlers, Matthias Rungger, Ayça Balkan, and Hadas Kress-Gazit. Automated generation of dynamics-based runtime certificates for high-level control. *Discrete Event Dynamic Systems*, pages 1–35, 2016.

[22] Jonathan DeCastro and Hadas Kress-Gazit. Synthesis of nonlinear continuous controllers for verifiably-correct high-level, reactive behaviors. *International Journal of Robotics Research*. Accepted.

[23] Jonathan A. DeCastro, Javier Alonso-Mora, Vasu Raman, Daniela Rus, and Hadas Kress-Gazit. Collision-free reactive mission and motion planning for multi-robot systems. In *Proceedings of the International Symposium on Robotics Research (ISRR)*, Sestri Levante, Italy, September 2015.

[24] Jonathan A. DeCastro and Hadas Kress-Gazit. Guaranteeing reactive high-level behaviors for robots with complex dynamics. In *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS 2013)*, 2013.

[25] Jonathan A. DeCastro and Hadas Kress-Gazit. Nonlinear controller synthesis and automatic workspace partitioning for reactive high-level behaviors. In *Proceedings of the 19th ACM International Conference on Hybrid Systems: Computation and Control (HSCC)*, Vienna, Austria, April 2016.

[26] Robin Deits and Russ Tedrake. Computing large convex regions of obstacle-free space through semidefinite programming. *Workshop on the Algorithmic Fundamentals of Robotics*, 2014.

[27] D. V. Dimarogonas, E. Frazzoli, and K.H. Johansson. Distributed event-triggered control for multi-agent systems. *IEEE Trans. Automatic Control*, 57(5):1291–1297, 2012.

[28] Jerry Ding, Jeremy Gillula, Haomiao Huang, Michael P. Vitus, Wei Zhang, and Claire J. Tomlin. Hybrid systems in robotics: Toward reachability-based controller design. *IEEE Robotics & Automation Magazine*, 18(3):33 – 43, Sept. 2011.

[29] Anca D. Dragan, Nathan D. Ratliff, and Siddhartha S. Srinivasa. Manipulation planning with goal sets using constrained trajectory optimization. In *IEEE International Conference on Robotics and Automation (ICRA 2011)*, pages 4582–4588. IEEE, 2011.

[30] R. Ehlers, R. Könighofer, and R. Bloem. Synthesizing cooperative reactive

mission plans. In *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, 2015.

[31] Rüdiger Ehlers. *Symmetric and Efficient Synthesis*. PhD thesis, Saarland University, 2013.

[32] Rüdiger Ehlers and Vasumathi Raman. Slugs: Extensible GR(1) synthesis. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, pages 333–339, 2016.

[33] Rüdiger Ehlers and Ufuk Topcu. Resilience to intermittent assumption violations in reactive synthesis. In *Proc. of the Int. Conf. on Hybrid Systems: Computation and Control*, 2014.

[34] Georgios E. Fainekos. Revising temporal logic specifications for motion planning. In *Proceedings of the IEEE Conference on Robotics and Automation*, 2011.

[35] Georgios E. Fainekos, Savvas G. Loizou, and George J. Pappas. Translating temporal logic to controller specifications. In *Proc. of the 45th IEEE Conf. on Decision and Control (CDC 2006)*, pages 899–904, 2006.

[36] Gerogios E. Fainekos, Antoine Girard, Hadas Kress-Gazit, and George J. Pappas. Temporal logic motion planning for dynamic robots. *Automatica*, 45(2):343 – 352, 2009.

[37] Cameron Finucane, Gangyuan Jing, and Hadas Kress-Gazit. Ltlmop: Experimenting with language, temporal logic and robot control. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, 2010.

[38] Emilio Frazzoli. *Robust hybrid control for autonomous vehicle motion planning*. PhD thesis, Massachusetts Institute of Technology, 2001.

[39] Antoine Girard, Giordano Pola, and Paulo Tabuada. Approximately bisimilar symbolic models for incrementally stable switched systems. *Automatic Control, IEEE Transactions on*, 55(1):116–126, 2010.

[40] G. Jing, R. Ehlers, and H. Kress-Gazit. Shortcut through an evil door: Optimality of correct-by-construction controllers in adversarial environments. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4796–4802, Nov 2013.

[41] A. Agung Julius, Georgios E. Fainekos, Madhukar Anand, Insup Lee, and George J. Pappas. Robust test generation and coverage for hybrid systems. In *Proc. of the 10th int. conf. on Hybrid systems: computation and control (HSCC'07)*, HSCC'07, pages 329–342, Berlin, Heidelberg, 2007. Springer-Verlag.

[42] S. Karaman and E. Frazzoli. Sampling-based motion planning with deterministic μ-calculus specifications. In *Proc. of the 48th IEEE Conf. on Decision and Control (CDC 2009)*, pages 2222–2229, 2009.

[43] Hassan K. Khalil. *Nonlinear Systems*. Prentice Hall, 3rd edition, 2002.

[44] D.E. Kirk. *Optimal Control Theory: An Introduction*. Prentice-Hall, 1976.

[45] Marius Kloetzer and Calin Belta. A framework for automatic deployment of robots in 2d and 3d environments. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2006, October 9-15, 2006, Beijing, China*, pages 953–958, 2006.

[46] Marius Kloetzer and Calin Belta. A fully automated framework for control of linear systems from ltl specifications. In *Proc. of the 9th Int. Conf. on Hybrid Systems: Computation and Control*, 2006.

[47] Marius Kloetzer and Calin Belta. Dealing with nondeterminism in symbolic control. In Magnus Egerstedt and Bud Mishra, editors, *Hybrid Systems: Computation and Control, 11th International Workshop (HSCC 2008)*, volume 4981 of *Lecture Notes in Computer Science*, pages 287–300. Springer, 2008.

[48] Ross A. Knepper and Daniela Rus. Pedestrian-inspired sampling-based multi-robot collision avoidance. In *RO-MAN*, pages 94–100. IEEE, 2012.

[49] Robert Könighofer, Georg Hofferek, and Roderick Bloem. Debugging formal specifications using simple counterstrategies. In *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009*, pages 152–159, 2009.

[50] Hadas Kress-Gazit, David C. Conner, Howie Choset, Alfred A. Rizzi, and George J. Pappas. Courteous cars. *IEEE Robot. Automat. Mag.*, 15(1):30–38, 2008.

[51] Hadas Kress-Gazit, Georgios E. Fainekos, and George J. Pappas. Translat-

ing structured english to robot controllers. *Advanced Robotics*, 22(12):1343–1359, 2008.

[52] Hadas Kress-Gazit, Gerogios E. Fainekos, and George J. Pappas. Temporal logic based reactive mission and motion planning. *IEEE Transactions on Robotics*, 25(6):1370–1381, 2009.

[53] Alex A. Kurzhanskiy and Pravin Varaiya. Ellipsoidal toolbox. Technical Report UCB/EECS-2006-46, EECS Department, University of California, Berkeley, May 2006.

[54] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, U.K., 2006. Available at http://planning.cs.uiuc.edu/.

[55] Wenchao Li, Lili Dworkin, and Sanjit A. Seshia. Mining assumptions for synthesis. In *9th IEEE/ACM Int. Conf. on Formal Methods and Models for Codesign, MEMOCODE*, 2011.

[56] Wenchao Li, Dorsa Sadigh, S. Shankar Sastry, and Sanjit A. Seshia. Synthesis for human-in-the-loop control systems. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014*, pages 470–484, 2014.

[57] J. Liu and N. Ozay. Abstraction, discretization, and robustness in temporal logic control of dynamical systems. In *Proc. of the 17th Int. Conf. on Hybrid Systems: Computation and Control (HSCC'14)*, 2014.

[58] Jun Liu, Necmiye Ozay, Ufuk Topcu, and Richard M. Murray. Synthesis of reactive switching protocols from temporal logic specifications. *IEEE Trans. Automat. Contr.*, 58(7):1771–1785, 2013.

[59] Jun Liu, Ufuk Topcu, Necmiye Ozay, and Richard M. Murray. Reactive controllers for differentially flat systems with temporal logic constraints. In *Proc. of the 51st IEEE Conf. on Decision and Control (CDC 2012)*, pages 7664–7670, 2012.

[60] Scott C. Livingston, Pavithra Prabhakar, Alex B. Jose, and Richard M. Murray. Patching task-level robot controllers based on a local μ-calculus formula. In *Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA)*, Karlsruhe, Germany, May 2013.

[61] Savvas G. Loizou and Kostas J. Kyriakopoulos. Automatic synthesis of

multiagent motion tasks based on ltl specifications. In *Proc. of IEEE Conf. on Decision and Control (CDC)*, 2004.

[62] Anirudha Majumdar and Russ Tedrake. Robust online motion planning with regions of finite time invariance. In *Proc. of the Workshop on the Algorithmic Foundations of Robotics*, 2012.

[63] Anirudha Majumdar, Mark Tobenkin, and Russ Tedrake. Algebraic verification for parameterized motion planning libraries. In *Proc. of the 2012 American Control Conference (ACC)*, 2012.

[64] MR Maly, M Lahijanian, Lydia E. Kavraki, H. Kress-Gazit, and Moshe Y. Vardi. Iterative temporal motion planning for hybrid systems in partially unknown environments. In *ACM International Conference on Hybrid Systems: Computation and Control (HSCC)*, pages 353–362, Philadelphia, PA, USA, 08/04/2013 2013. ACM, ACM.

[65] Manuel Mazo, Anna Davitian, and Paulo Tabuada. Pessoa: A tool for embedded controller synthesis. In *22nd International Conference on Computer Aided Verification (CAV 2010)*, pages 566–569, 2010.

[66] A. Megretski. Systems polynomial optimization tools (spot). http://web.mit.edu/ameg/www/.

[67] Daniel Mellinger, Aleksandr Kushleyev, and Vijay Kumar. Mixed-integer quadratic program trajectory generation for heterogeneous quadrotor teams. *Robotics and Automation, IEEE Int. Conf. on*, 2012.

[68] Petter Nilsson and Necmiye Ozay. Incremental synthesis of switching protocols via abstraction refinement. In *53rd IEEE Conference on Decision and Control, CDC 2014, Los Angeles, CA, USA, December 15-17, 2014*, pages 6246–6253, 2014.

[69] Jerome Le Ny and George J. Pappas. Sequential composition of robust controller specifications. In *IEEE International Conference on Robotics and Automation (ICRA 2012)*, pages 5190–5195, 2012.

[70] Giuseppe Oriolo, Alessandro De Luca, Ro De Luca, and Marilena Vendittelli. Wmr control via dynamic feedback linearization: Design, implementation, and experimental validation. *Control Systems Technology, IEEE Transactions on*, 10(6):835–852, November 2002.

[71] G. Pola, A. Girard, and P. Tabuada. Approximately bisimilar symbolic models for nonlinear control systems. *Automatica*, 44(10):2508–2516, 2008.

[72] Vasumathi Raman. Reactive switching protocols for multi-robot high-level tasks. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2014)*, pages 336–341, 2014.

[73] Vasumathi Raman and Hadas Kress-Gazit. Towards minimal explanations of unsynthesizability for high-level robot behaviors. In *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS 2013)*, 2013.

[74] Vasumathi Raman and Hadas Kress-Gazit. Synthesis for multi-robot controllers with interleaved motion. In *IEEE International Conference on Robotics and Automation (ICRA 2014)*, pages 4316–4321, 2014.

[75] Vasumathi Raman, Nir Piterman, and Hadas Kress-Gazit. Provably correct continuous control for high-level robot behaviors with actions of arbitrary execution durations. In *IEEE Int. Conf. on Robotics and Automation*, 2013.

[76] G. Reißig. Computing abstractions of nonlinear systems. *IEEE Transactions on Automatic Control*, 56(11):2583–2598, 2011.

[77] Indranil Saha, Rattanachai Ramaithitima, Vijay Kumar, George J. Pappas, and Sanjit A. Seshia. Implan: Scalable incremental motion planning for multi-robot systems. In *Proceedings of the 7th International Conference on Cyber-Physical Systems (ICCPS)*, April 2016.

[78] Philipp Schillinger, Mathias Bürger, and Dimos V. Dimarogonas. Decomposition of finite LTL specifications for efficient multi-agent planning. In *13th International Symposium on Distributed Autonomous Robotic Systems*, Springer Tracts in Advanced Robotics, 2016.

[79] Paulo Tabuada and George J. Pappas. Linear time logic control of discrete-time linear systems. *IEEE Trans. Automat. Contr.*, 51(12):1862–1877, 2006.

[80] Ryo Takei, Haomiao Huang, Jerry Ding, and Claire J. Tomlin. Time-optimal multi-stage motion planning with guaranteed collision avoidance via an open-loop game formulation. In *IEEE International Conference on Robotics and Automation (ICRA 2012)*, pages 323–329, 2012.

[81] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[82] Russ Tedrake, Ian R. Manchester, Mark Tobenkin, and John W. Roberts. Lqr-trees: Feedback motion planning via sums-of-squares verification. *I. J. Robotic Res.*, 29(8):1038–1052, 2010.

[83] Jana Tumova and Dimos V. Dimarogonas. Decomposition of multi-agent planning under distributed motion and task LTL specifications. In *54th IEEE Conference on Decision and Control, CDC 2015, Osaka, Japan, December 15-18, 2015*, pages 7448–7453. IEEE, 2015.

[84] Jana Tumova, Boyan Yordanov, Calin Belta, Ivana Cerna, and Jiri Barnat. A symbolic approach to controlling piecewise affine systems. In *49th IEEE Conference on Decision and Control (CDC)*, pages 4230 –4235, 2010.

[85] Alphan Ulusoy, Michael Marrazzo, and Calin Belta. Receding horizon control in dynamic environments from temporal logic specifications. In *Robotics: Science and Systems*, 2013.

[86] Alphan Ulusoy, Stephen L. Smith, Xu Chu Ding, Calin Belta, and Daniela Rus. Optimality and robustness in multi-robot path planning with temporal logic constraints. *I. J. Robotic Res.*, 32(8):889–911, 2013.

[87] J van den Berg, S J Guy, M Lin, and D Manocha. Reciprocal n-body Collision Avoidance. In *Int. Symp. on Robotics Research (ISRR)*, 2009.

[88] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for concurrency*, pages 238–266. Springer, 1996.

[89] Eric M. Wolff, Ufuk Topcu, and Richard M. Murray. Automaton-guided controller synthesis for nonlinear systems with temporal logic. In *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS 2013)*, 2013.

[90] Kai Weng Wong, Rüdiger Ehlers, and Hadas Kress-Gazit. Correct high-level robot behavior in environments with unexpected events. In *Proc. of Robotics: Science and Systems*, June 2014.

[91] T. Wongpiromsarn, U. Topcu, and R.M. Murray. Receding horizon temporal logic planning. *Automatic Control, IEEE Transactions on*, 57(11):2817–2830, 2012.

[92] T. Wongpiromsarn, A. Ulusoy, C. Belta, E. Frazzoli, and D. Rus. Incremental synthesis of control policies for heterogeneous multi-agent systems

with linear temporal logic specifications. In *Robotics and Automation (ICRA), IEEE Int. Conf. on*, 2013.

[93] Tichakorn Wongpiromsarn, Ufuk Topcu, and Richard M. Murray. Receding horizon control for temporal logic specifications. In *Proc. of the 13th Int. Conf. on Hybrid Systems: Computation and Control (HSCC'10)*, 2010.

[94] B. Yordanov, J. Tumova, I. Cerna, J. Barnat, and C. Belta. Temporal logic control of discrete-time piecewise affine systems. *Automatic Control, IEEE Transactions on*, 57(6):1491–1504, 2012.

[95] M. Zamani, G. Pola, M. Mazo, and P. Tabuada. Symbolic models for nonlinear control systems without stability assumptions. *IEEE Transactions on Automatic Control*, 57(7):1804–1809, 2012.